Alejandro Fernandez UFID: 2265-5282

OpenGL Viewer Report

Introduction

In this assignment, an OpenGL model viewer was created by extending the model transform applications from the previous assignment. The main changes made were the addition of a perspective projection matrix and various shaders. There was a total of five shader programs implemented: Z-buffer, Z'-buffer, Gouraud shading, Phong shading, and flat shading shaders. Through the utilization of the ImGui library, the application gives a user-friendly way to manipulate the perspective projection matrix and swap between shaders.

Perspective Projection Matrix

A perspective projection matrix is applied to an object in view space to give it the illusion of three-dimensionality. The transformation does this by skewing the view space to make lines seem like their converging toward a horizon line. To get this matrix, a function within the Handmade Math library was used. This function takes in a field of view, aspect ratio, near plane, and far plane as its parameters.

$$\boldsymbol{P} = \begin{bmatrix} \frac{1}{aspect\ ratio * \tan\left(\frac{fov}{2}\right)} & 0 & 0 & 0 \\ u_x & \frac{1}{\tan\left(\frac{fov}{2}\right)} & 0 & 0 \\ f_x & f_y & \frac{n+f}{n-f} & \frac{2*n*f}{n-f} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

In this above matrix the aspect ratio is the ratio of the width and height of the window the scene is being displayed to. The field of view is the internal angle between the two lines that go from the origin of the camera to the horizontal extent of the far plane. The near and far plane describe the dimensions of the box defined by this matrix. Any object closer than the near plane or farther than the far plane will not be rendered to the screen. The application UI gives the user the ability to manipulate the field of view, near plane, and far plane directly. The aspect ratio can be modified indirectly by changing the size of the scene window.

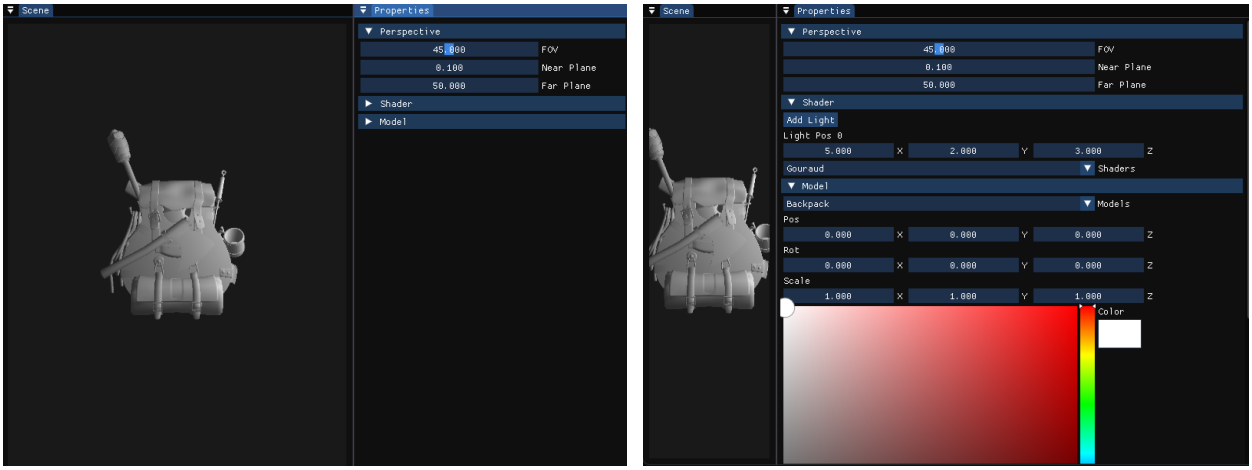

Figure 1 – Highlights modification of aspect ratio. Decreasing the width of the scene view indirectly modifies the aspect ratio, keeping the resulting image from squishing.

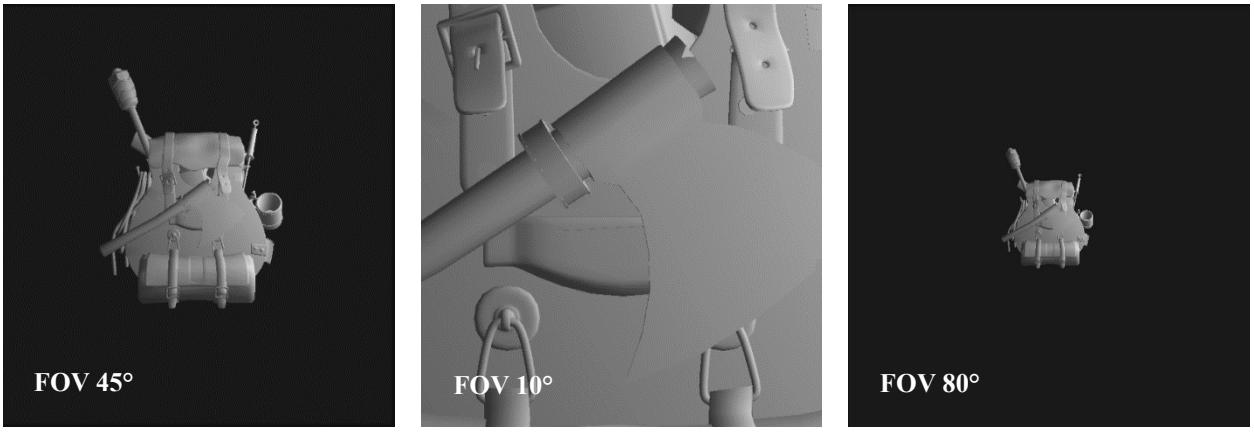

Figure 2 – Highlights modification of field of view. Decreasing the field of veiw creates a zoom in effect whilst increasing the field of view creates a zoom out effect.

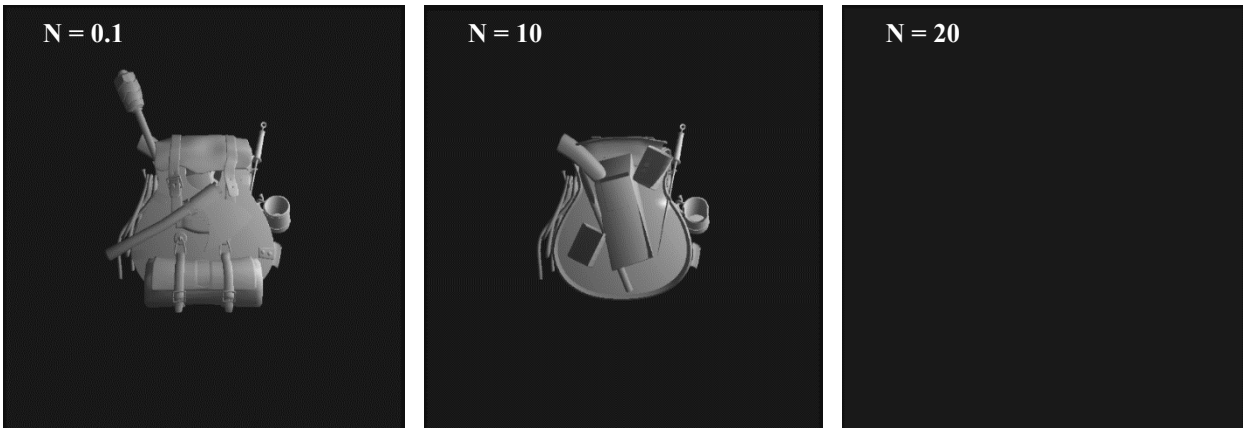

Figure 3 – Highlights modification of the near plane. As the near plane value increases, the point at which objects start clipping seems to be farther. With a small clipping value, you can see objects that are close; however, as it increases, those objects become less visible until they are completely clipped away.

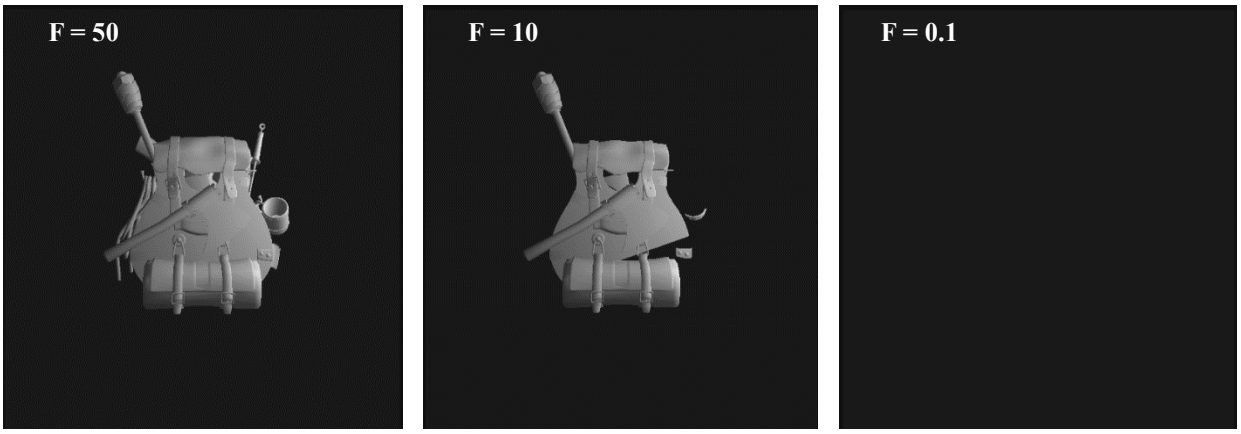

Figure 4 – Highlights modification of the far plane. As the near plane value decreases, the point at which objects start clipping seems to be closer. With a larger clipping value, you can see objects that are far away; however, as it decreases, those objects become less visible until they are completely clipped away.

Depth Buffers

The Z and Z'-buffer contain depth values for each pixel, with each value corresponding to the object closest to the camera. OpenGL creates this buffer by default, so these depth values only need to be accessed within the shader and displayed on the screen. Outputting the raw depth values will give the Z-buffer. The depth value within each pixel ranges from 0 to 1 and expresses an interpolation of the closest object position between the near and far planes. The problem with this raw value is that they are non-linear; more specifically, they are proportional to 1/z. This creates very little distinction amongst the depth values of objects near the camera.

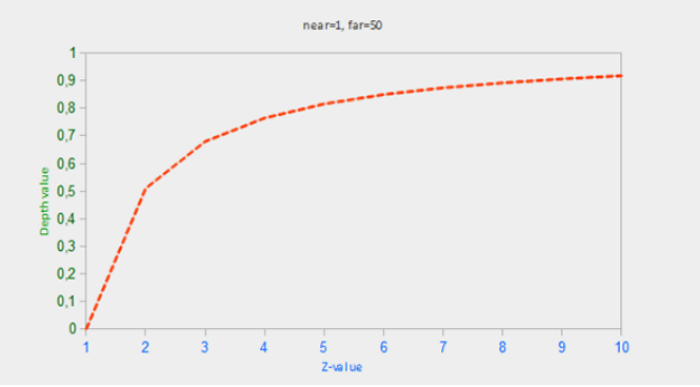

Figure 5 (1) – Shows non-linearity of z values due to projection.

To overcome this issue, the depth values must be linearized. This linearization of the depth values will allow clear distinctions amongst the depth values no matter their distance from the camera. The linearization is done by converting the projected z value back to the z value in view space and dividing it by the far plane value to get a number between 0 and 1.

$$z_e = \frac{2 * n * f}{f + n - z_n * (f - n)}$$

The above equation highlights the relationship between the z values in view space and the z value after projection. Applying this equation to the shade will linearize the z values, which can be seen in the example images below.

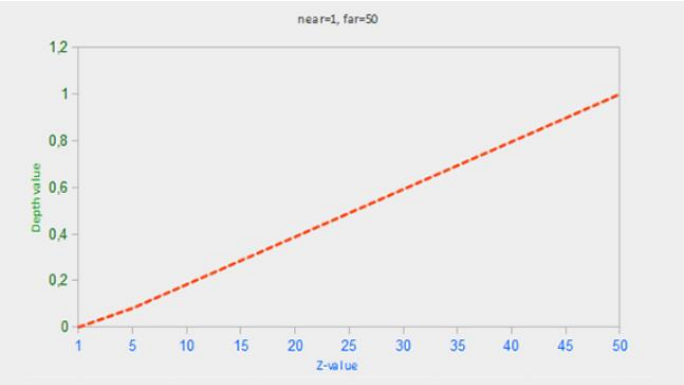

Figure 6 (1) – Shows z-values after linearization. There is more precision for smaller z-values.

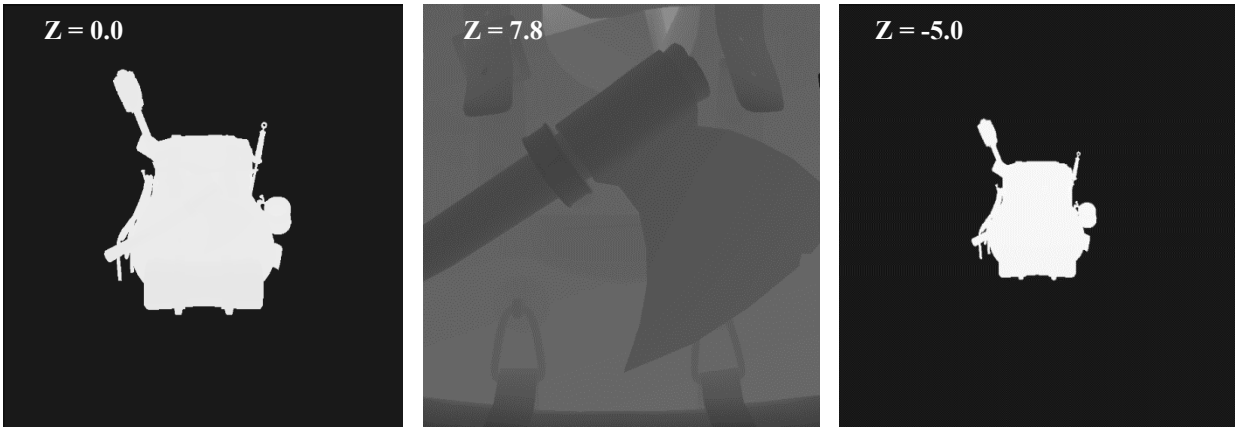

Figure 7 – Z-buffer with object at different distances from the camera. The lack of precision can be scene by how getting farther away from the object doesn't seem to change the sceen. The object also has to get very close to change the buffer in a significant way. The near and far plane values are 1.5 and 20, respectively.

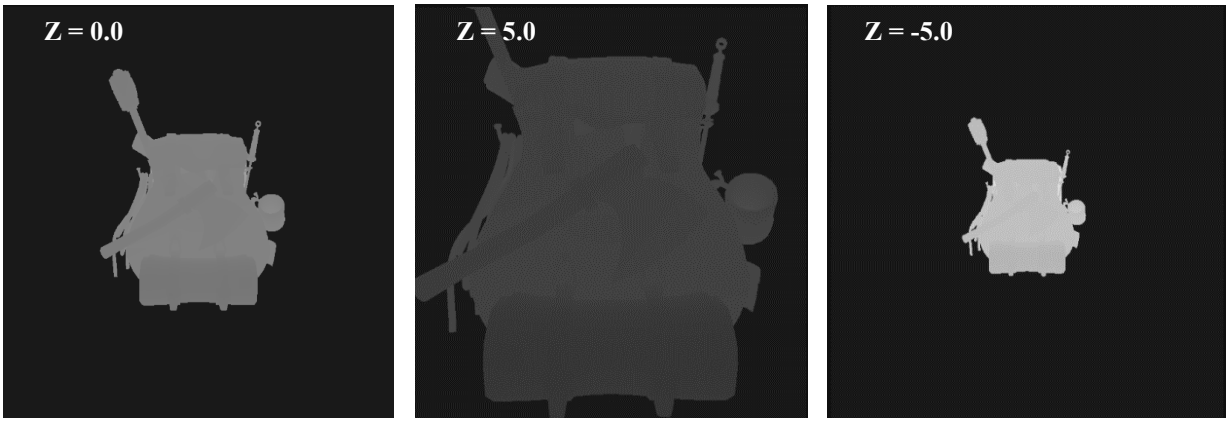

Figure 8 – Z'-buffer with object at different distances from the camera. Changing the distance creates a smoother transition in the z-value. This allows for a clearer distinction of the distance of the model. The near and far plane values are 1.5 and 20, respectively.

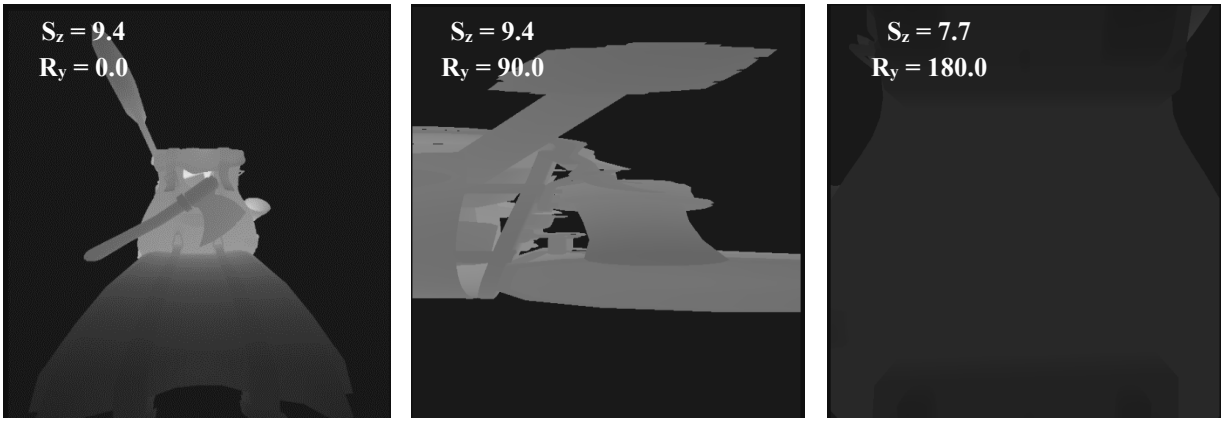

Figure 9 – Z'-buffer with object a scale applied to it and different orientations. The sections of the object that are closer are darker than those that are farther. This shows that the buffer is consistent with transformations.

Gouraud and Phong Shading

To implement lighting within this model viewer, Gouraud and Phong shading were implemented to calculate the color of each pixel. These two methods are essentially the same but are executed in different sections in the graphics pipeline. Gouraud shading is calculated per vertex whilst Phong shading is calculated per fragment. This makes Gouraud shading a lot more efficient; however, the quality is worse.

Other than the differences mentioned previously, the lighting calculation for both methods is the same. The light, view, and reflected direction are calculated and used alongside the material properties to get diffuse and specular shading. The equation below includes the

normal at the vertex/fragment ($\hat{N}$), light direction ($\hat{L}$), view direction ($\hat{V}$), reflection direction ($\hat{R}$), diffuse coefficient (k_d), specular coefficient (k_s), and ambient coefficient (k_a).

$$I_d = \max\left(\hat{N} \cdot \hat{L}\right), 0) * k_d$$

$$I_s = \max\left(\hat{R} \cdot \hat{V}, 0\right)^n * k_s$$

$$I_a = k_a$$

Flat Shading

Following the trend of minimizing computation time alongside quality, flat shading attempts to do a shading calculation for triangles. This is done by running the lighting calculation on the triangle and setting the entire triangle to the resulting color. Due to all the models having their normals averaged out in the OBJ file, the normal will need to be recalculated.

This can be elegantly solved by executing the shading calculation in the geometry shader. Since this shader works with triangles, it gives access to the vertices that make up each triangle. This allows for the calculation of the normal of the triangle's surface by taking the cross product of two segments of the triangle.

Other than this, similar to the two previous examples, the lighting calculation is the same. The only significant difference is that the center of the triangle is used as the position rather than the vertices or fragments.

Figure 10 – Show monkey model shaded using Gouraud shading (left), Phong shading (middle), and Flat shading (right). Light positon is (7.5, 5.0, 4.5).

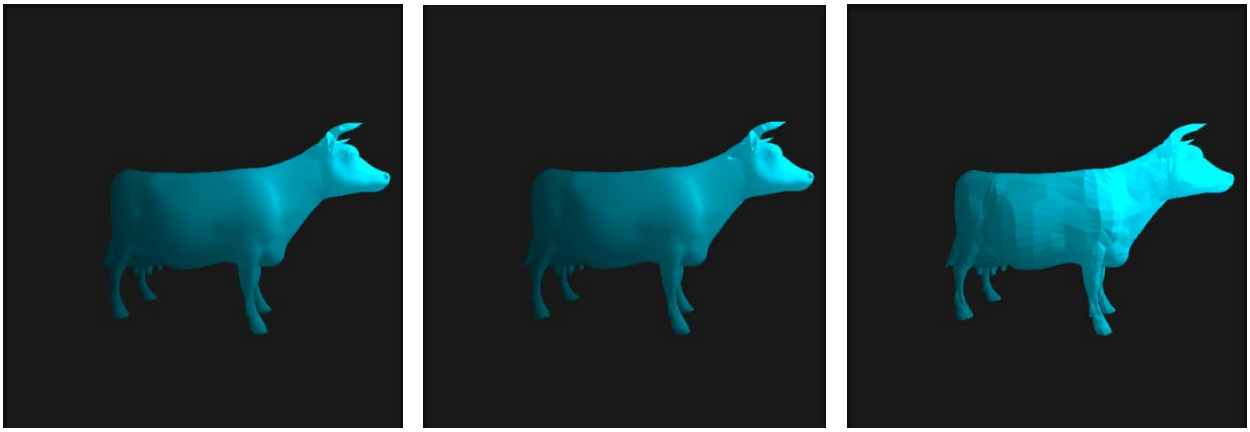

Figure 11 – Show cow model shaded using Gouraud shading (left), Phong shading (middle), and Flat shading (right). Light positon is (7.5, 5.0, 4.5).

The main differences that comes from each shading technique can be pinpointed through observing the way they each execute their shading calculation. Due to Gourand shading being applied per vertex, there is some details that are missed. This becomes more apprent when the number of triangles that make up a model is small. Phong shading, on the other hand, has a tremendous amount of precision, since it executes that shading calculation per fragment. Flat shading, as the name implies, is the least detailed of the bunch because it is ran per triangle. This leads to a discontinious surface, with a lot of jaged edges.

References

(1) *LearnOpenGL - Depth testing*. (n.d.). Learnopengl.com. https://learnopengl.com/Advanced-OpenGL/Depth-testing

Video Link : https://youtu.be/ZkNBYdf30SI