Assignment 3: Viewing Transformations and Shading

By

Duncan Stewart, Sai Ravi Teja Gangavarapu

UFID Duncan Stewart: 1644-7760

UFID Sai Ravi Teja Gangavarapu: 1050-4370

1 Modelview Transformations

Starting out we needed to perform adding the Perspective Matrix as stated in the homework document. At first we added the performance matrix right after matrices like that of scaling and transforming so for organization wise we can know where to look while also being able to immediately initialize the matrix after we have read the file that we are currently reading. In a lot of our examples we used the pig obj to document this as it was more helpful that way to document. The statement is provided below:

glm::mat4 perspectiveMatrix = glm::perspective(fov, aspectRatio, nearPlane, farPlane);

In order to get some of the values that you can see above like that of *fov, aspectRatio, nearPlane,* and *farPlane* we had to make statements for each describing the process of how to get them. First off is the field of view or in our case the *fov* which will be found through the statement below:

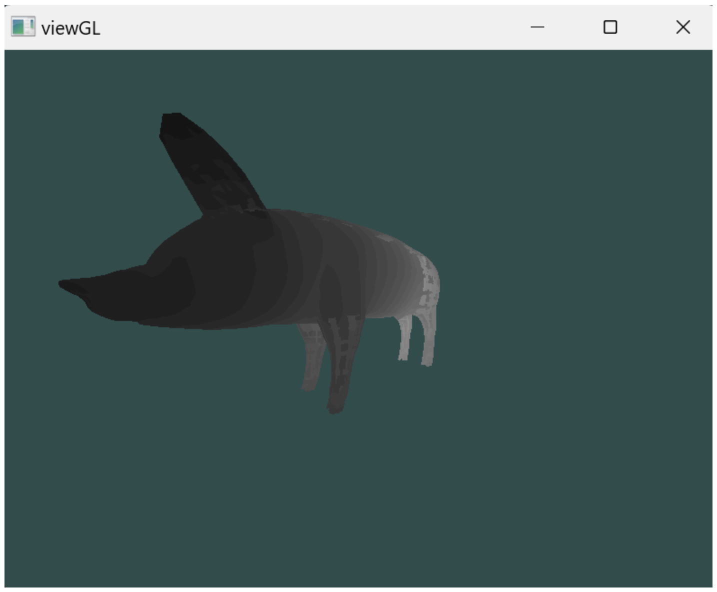
float fov = glm::radians(120.0f)

After this we go down to the *aspectRatio* which will be easily provided by dividing the width and the height of the overall window we are using which will be calculated by depending on the screen you are using at that moment. However, the next two variables are two that are set by what we want them to be so in the predetermined case it would be that of:

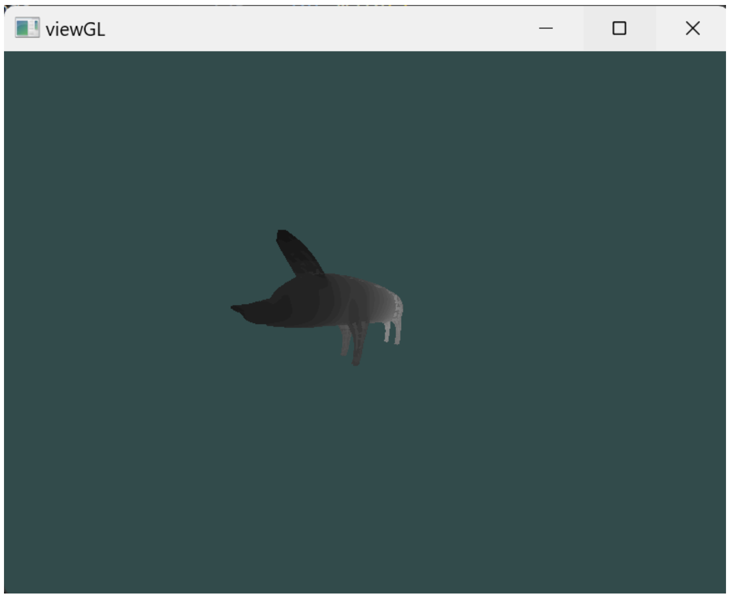
float nearPlane = 0.1f;

float farPlane = 3.0f;

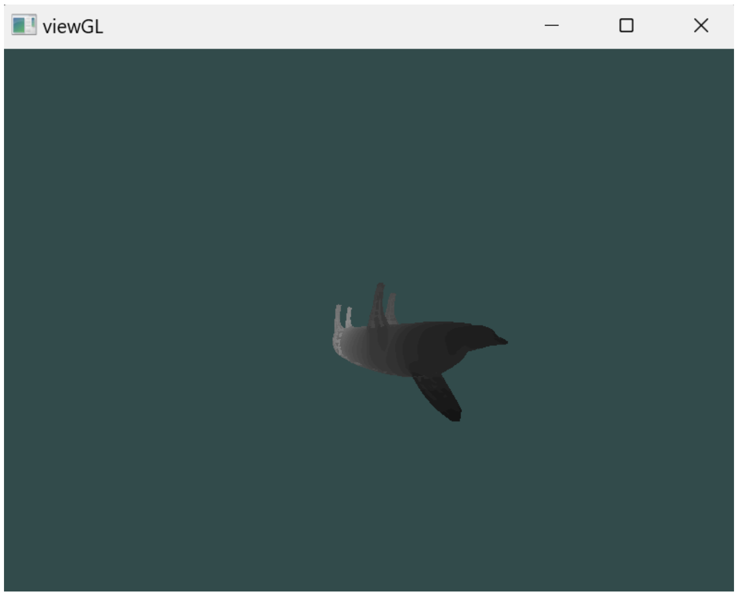
What these two values give us is the clipping plane distance so as expected from near to far. However, to see what changes when we change the different values within the perspective matrix, the values of *nearPlane, farPlane,* and *fov.* The different type of experiments are shown below:



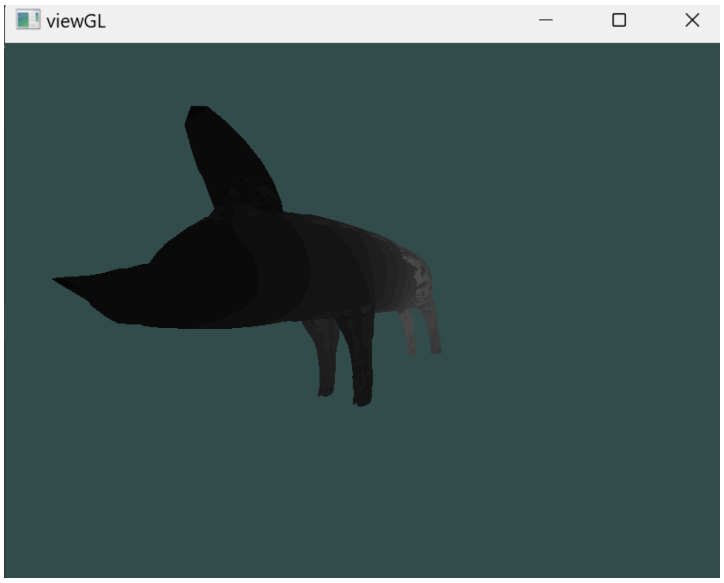
fov: 2.1 radians
nearPlane: 0.1
farPlane: 3



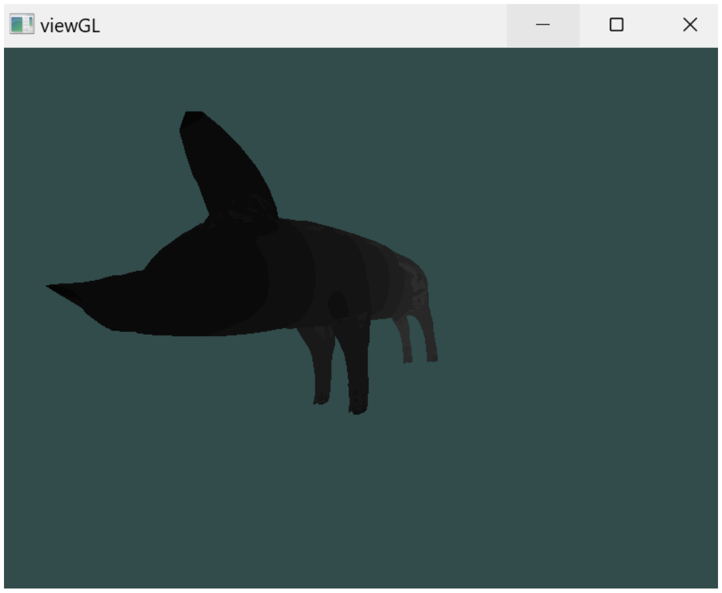
fov: 2.64 radians
nearPlane: 0.1
farPlane: 3



fov: -2.64 radians
nearPlane: 0.1
farPlane: 3



fov: 2.1 radians
nearPlane: 0.27
farPlane: 3



fov: 2.1 radians
nearPlane: 0.27
farPlane: 19.95

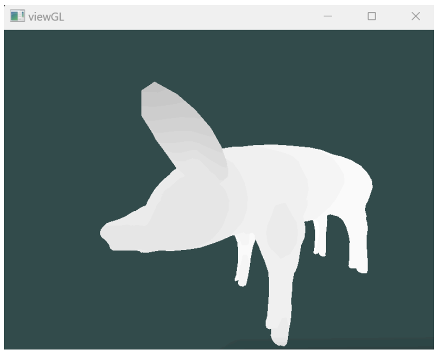
The initial statement we saw earlier in this section providing what the *perspectiveMatrix* will be is once again provided at the end of our code in ebo.cpp which will allow it to be sure to have it updated with all of the information that has been provided throughout the code and changes that may have come from it.

2 Z-Buffer

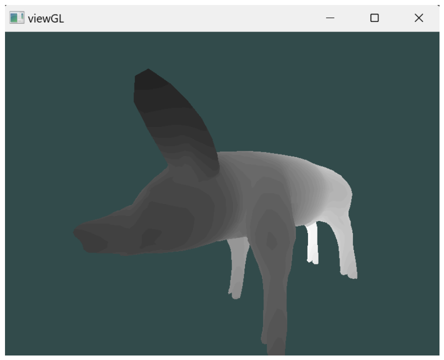
When starting off on the z-buffer we needed to have a depth test implemented as to the true depth of the z-buffer within the code. In the beginning of the code on file *ebo.cpp* we have to initialize and enable the depth test that we have from OpenGL. With the initial enabling from the code provided below:

glEnable(GL_DEPTH_TEST)

This code would be used later on to also load the depth test and other assortment of OpenGL tools that we used within our program.



z-buffer



z'-buffer

2.1 Source.fs

Within the *Source.fs* file we calculate the depth that is going into the depth test that we discussed earlier. Starting from the *perspectiveMatrix* values of *nearPlane* and *farPlane* are used within our function *LinearizeDepth.* So depending on the values set during the perspective matrix results could vary but for the base values we are still using near as 0.1 and far as 3.0. After doing this we head into our previously discussed function where we will get the value of z. The statement to get z is provided below:

*float z = depth * 2.0 - 1.0*

After this we use the value we found for z to calculate depth outside of the current function with the statement:

*(2.0 * near * far) / (far + near - z * (far - near))*

With this being returned to the main function within the *Source.fs* file we will use the returned value to provide the depth we need as shown below:

float depth = LinearizeDepth(gl_FragCoord.z) / far

Since getting the depth, we can now use that to get the fragmented color depending on the depth that was received which once again will depend on many factors but also include that of the *perspectiveMatrix.* We will get the fragmented color with the statement below:

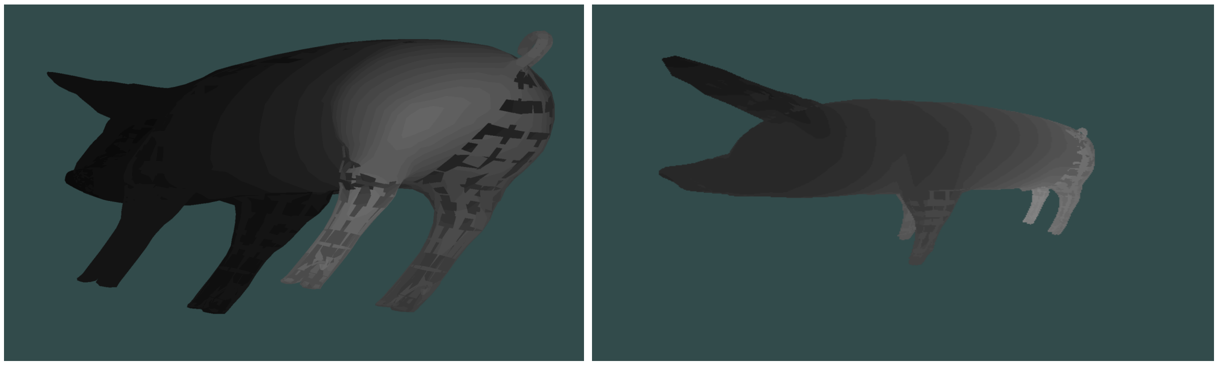
FragColor = vec4(vec3(depth), 1.0)

2.2 Rotating/ Scaling the Scene

To be able to rotate or scale the scene to be able to look at the depth of the scene we had to use former techniques like using WASD keys to translate the scene while also using other keys to move around the scene and allow us to see even more detail of the overall scene. XYZ keys were used to rotate around the different axis they represent while F and G keys would have you change the *fov* variable depending on what you would like the scene to look like. The LEFT and RIGHT keys were used to change far and near plane variables so once again you could change different variables while the scene was already currently open. Lastly the UP and DOWN keys were used to change the scene by doing as shown below:

```
sX *= 1.01;
sY *= 1.01;
sZ *= 1.01;

sX *= 0.99;
sY *= 0.99;
sZ *= 0.99;
```



Examples of rotating and scaling the scene with the help of z-buffers for depth

3 Gouraud and Phong Shading

The two types of shading present through this assignment is that of Gouraud and Phong shading which as in class and other info online the main difference between the two shaders can be seen as "When the Phong lighting model is implemented in the vertex shader it is called Gouraud shading instead of Phong shading." To showcase this difference between the two types of shaders we will go through them individually and how we implemented this throughout our code.

3.1 Phong Shading

We started off with Phong as it would be easier to implement Gouraud as putting Phong into the vertex shader supposedly would do the trick. However, starting off with Phong we made two separate files, *Phong.fs* and *Phong.vs*, where we will do our calculations for the shading on the different objects we tested. Starting off in *Phong.vs*, we need to get the fragment position and the normals from the object we are using for that current example. How we achieve these values is by using the different matrices we have from *ebo.cpp* which can be seen below:

```
FragPos = vec3(modelMatrix * vec4(aPos, 1.0));
Normal = mat3(transpose(inverse(modelMatrix))) * aNormal;
```

After this we go to *Phong.fs* where the shader needs ambient shading and is calculated by, where *ambientStrength* is set at 0.3 for base trial:

```
vec3 ambient = ambientStrength * lightColor
```

After this has been completed we will head to diffuse shading where we will need the *norm, lightDir,* and *diff.* These can be provided by the normals provided by the previous calculations and then normalizing the light position subtracted by the fragment position which would give us the light direction. Lastly, we use the last two variables by getting the max of the dot product of them to then give us *diff.* After all of this is done we can now have our diffuse shading with the equation:

```
vec3 diffuse = diffuseStrength * diff * lightColor
```

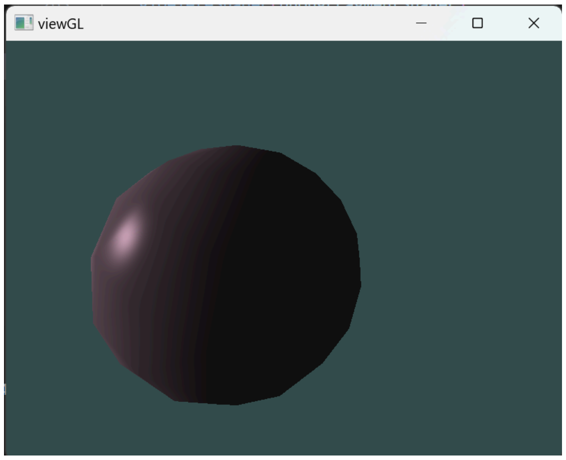
Lastly, we come to specular shading where we start off with getting the view Direction which will be achieved through normalizing the camera position subtracted by the fragment position. After this has been retrieved we get the reflected direction through reflecting off the negative of the light direction and the normals provided. Lastly we achieve specular shading through the last two statements below:

```
float spec = pow(max(dot(viewDir, reflectDir), 0.0), 64);
```

*vec3 specular = specularStrength * spec * lightColor;*

After all of this has been achieved we will add up our three different shaders multiplied by the object color to get our result which will be used in the overall fragment color.

Once we have done all of these calculations we have now gotten our Phong shading which can be described as shining back towards the camera now matter where you look as the fragment's have been calculated to get the light from wherever it was shining. As we can see from the images below the result is probably the closest to what it should like while also still can be described as a bit blocky as you get closer with the camera.



Phong Shading on Sphere

3.2 Gouraud Shading

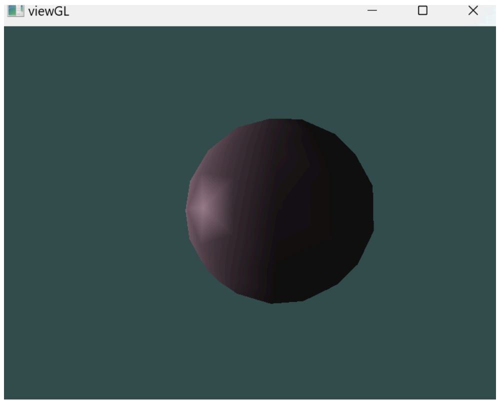
With Gouraud shading it is essentially the same thing as Phong shading before it however the main difference is that where we do all the calculations is that of the vertex shader file labeled as *Gouraud.vs.* With the fragment shader only having the single statement in the main function as seen below:

FragColor = vec4(vertexColor, 1.0)

As seen before with the Phong shading we would go down and calculate the different shaders as before but now in the vertex shader as that is the main difference between the two shaders. However, the last equation which will provide the vertex color rather than the fragment color:

*vertexColor = (ambient + diffuse + specular) * objectColor*

The difference in look can be seen from that of the reflection which has a hexagonal figure rather than the circle reflection from the light we saw in the Phong shading. So there is a massive difference in look as can be seen in the image below:



Gouraud Shading on Sphere

4 Flat Shading

The initial difference of flat shading to that of the other shaders that have been presented in this assignment is how flat shading will be loading in the object in our file *objLoader.cpp.* The other shaders loaded the object in as normal and as can be seen in the various vertex shaders and fragment shaders there is a function dedicated to loading the object for flat shading. The difference of loading in the objects between the normal way seen throughout the program and that of the flat shading is that of the use of vertices and normals. We use these normals and vertices to attach themselves to the obj so that we can load the object in like triangles, with each triangle having its own color. The normals are added into an index of sorts so we know which normal is exactly attributed to which place on the object as this is important for the shading to know what color to put on that specific fragment.

```
for (size_t j = 0; j < 3; ++j) {

const auto& index = shape.mesh.indices[i + j];

normal.x += attrib.normals[3 * index.normal_index + 0];

normal.y += attrib.normals[3 * index.normal_index + 1];

normal.z += attrib.normals[3 * index.normal_index + 2];

}

normal = glm::normalize(normal);
```

Their vertices are all pushed back into a vector so that we can later use these vertices for use in shading.An example of how we end up using these vertices is shown in the function itself to calculate the average normal to all vertices to a face on the object. The statements to get said result are shown below:

```
const auto& index = shape.mesh.indices[i + j];

vertices.push_back(attrib.vertices[3 * index.vertex_index + 0]);

vertices.push_back(attrib.vertices[3 * index.vertex_index + 1]);

vertices.push_back(attrib.vertices[3 * index.vertex_index + 2]);

vertices.push_back(normal.x);
```

vertices.push_back(normal.y);

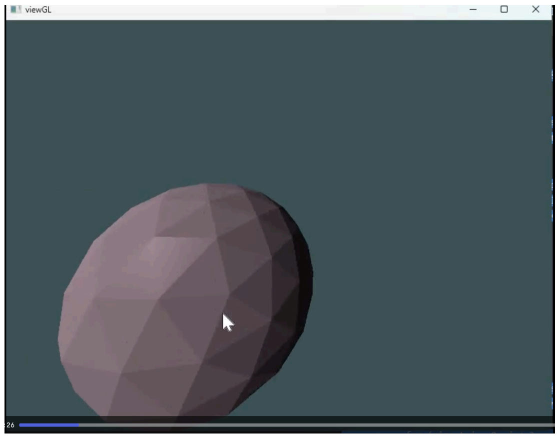
vertices.push_back(normal.z);

After this loading in we go to the flat files. We have two files once again of *flat.fs* and *flat.vs.* Starting off with the vertex shader it is similar in design to the *Phong.vs* file, where it provides the fragment color, normals and our color for the object. Once we get to the main function of the fragment shader we begin to use the ambient shading with the ambient strength. Once this is done we start to get the diffuse shading which we will provide by normalizing the normals and the light position subtracted by the fragment position. Once we have gotten the diffuse shading we will then head to the specular which is of similar caliber to that of diffuse but rather we will focus on the reflection and the direction at which that comes so there will be a difference of color even though flat shading has a tendency to not blend well together. After doing all of the shading we can then get the result which will result in:

*vec3 result = (ambient + diffuse + specular) * objectColor;*

FragColor = vec4(result, 1.0);

After this is done we will receive a result like that of what can be seen below, where it is much more block and is fragmented in the way that each fragment is colored a different way rather than them truly blending in together as we talked about in class.



Flat Shading on Sphere