HW #2: Model Transformations

Joshua Labasbas
UFID: #3766 3960
Joshua.Labasbas@ufl.edu

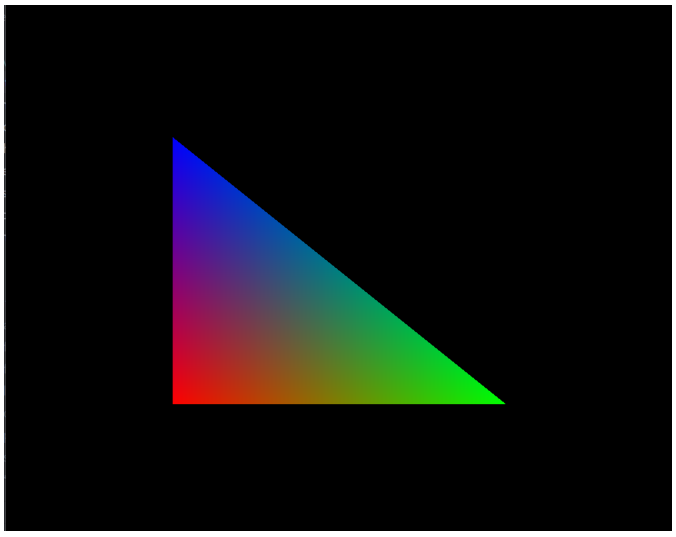
March 9, 2024

1 Task #1

The first task was to change the color of each vertex of the base triangle. The task was acheived by modifying the VBO and the VAO. I used the following table for my vertex buffer data.

Vertex	x	y	z	r	g	b
1	-0.5	-0.5	0.0	1.0	0.0	0.0
2	0.5	-0.5	0.0	0.0	1.0	0.0
3	-0.5	0.5	0.0	0.0	0.0	1.0

Then for the VAO I added another attribute pointer for the color attribute. Then from the shader side, I took in the position vector and added a fourth w dimension to the position vector for th vertex shader. Then for the fragment shader, it takes in the color from the vertex shader then outputs the same color. These values were hard coded in since we are only dealing with one triangle. Resulting in the final image:



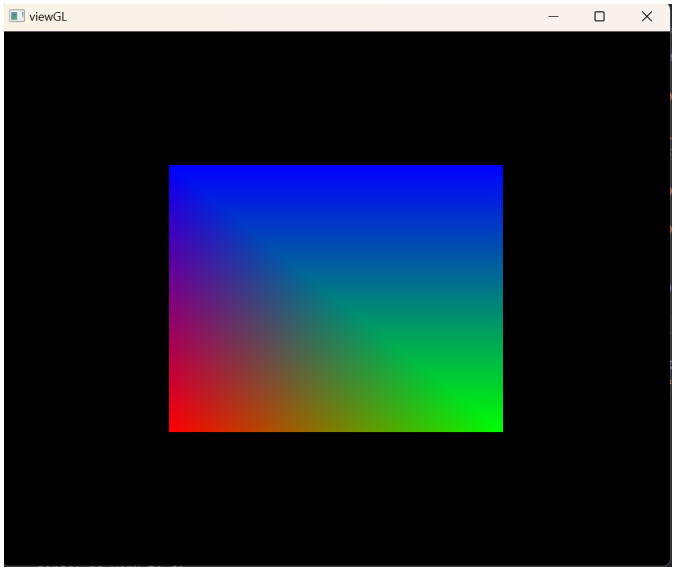
2 Task #2

The next task was to read the .obj files into vertices and then loading them into a data structure to be rendered. I did this first by using the pawn.obj file since it is made of purely triangles. Then I read in the file using the string streams, first I read in all the vertices and make them into the vertex object. Then from there I read in the indices in the same way. These indicies are then compiled into the triangles struct, therefore, I use an indexed triangles data structure instead of the separate triangles data structure. This was because it seemed like the best way that the data in the .obj file presented itself. In the case of the simple single triangle in the first task but with two triangles to make a rectangle, the data looks like the following:

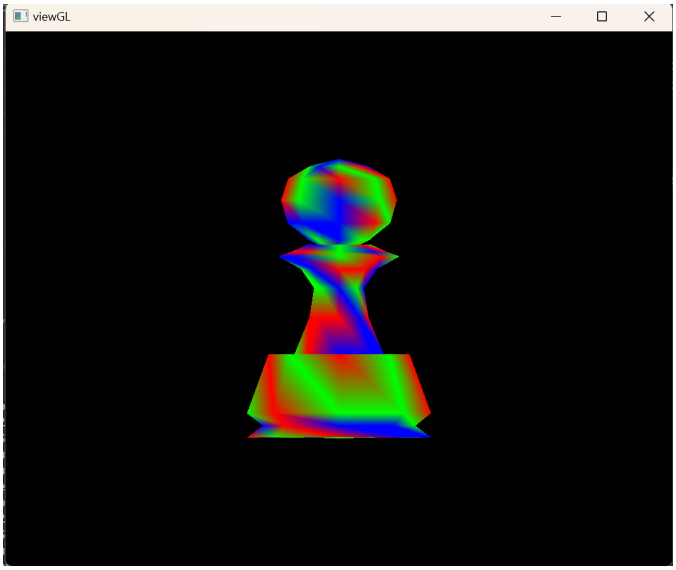
Vertex	x	y	z	r	g	b
0	-0.5	-0.5	0.0	1.0	0.0	0.0
1	0.5	-0.5	0.0	0.0	1.0	0.0
2	-0.5	0.5	0.0	0.0	0.0	1.0
3	0.5	0.5	0.0	0.0	0.0	1.0

Triangle	Index 1	Index 2	Index 3
1	0	1	2
2	2	1	3

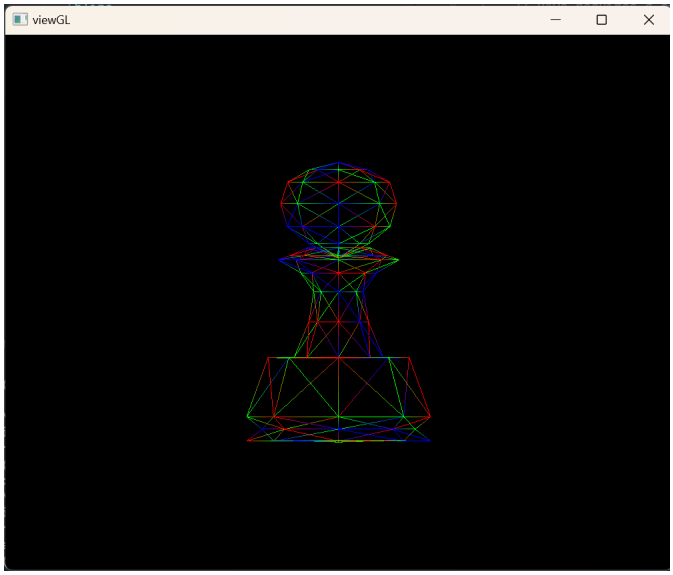
This winds up looking like the following:



This process is repeated across the pawn.obj file to result in the following image:



And then with wireframe:



3 Task #3

This task asks for the shader code to be moved from const char * to separate files. The way I did this was by reading in the file and each file was labeled with "#shader shader/fragment", this line marks the type of shader being read in. Then the application reads in each line then pipes the line into the char buffer with a new line character at the end of each line. That process recreates the char pointer that we had initially operated with with the simple single triangle.

4 Task #4

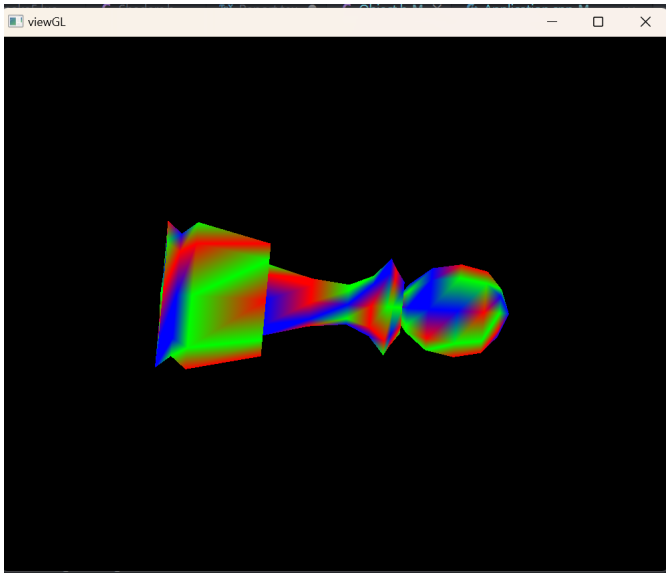
In this task I had to implement model transformations. Following LearnOpenGL.com, the first method was that of sending a transformation matrix to the shader to calculate on the GPU. Pressing the WASD keys control rotating the pawn on various axis', the arrow keys contol the translation transformation, and Q and E controls the scaling of the pawn. There is a central model matrix in the Object class, and the Object class has methods that apply transformations to the Object transformation matrix, these methods taking in different axis' as arguements.

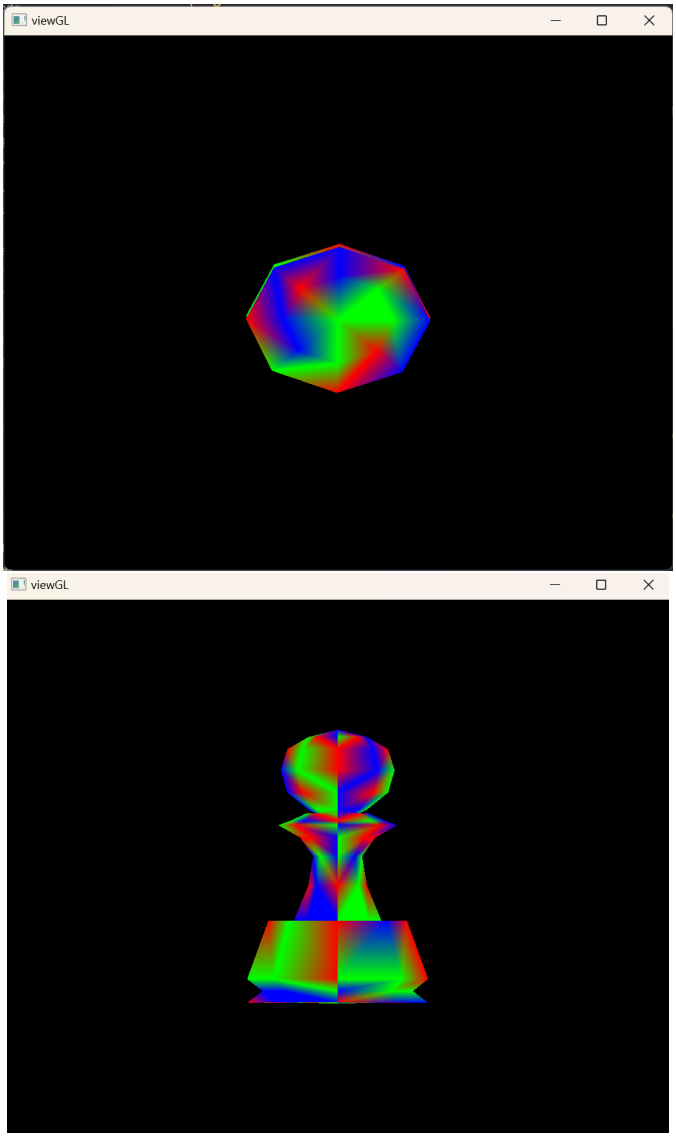
4.1 Rotation

The following table shows the mapping between keys and axis'

Key	Axis
A	(0.0, 0.0, 1.0)
D	(0.0, 0.0, -1.0)
W	(1.0, 0.0, 1.0)
S	(-1.0, 0.0, 0.0)
R	(0.0, 1.0, 0.0)
T	(0.0, -1.0, 0.0)

The reversed axis allows for setting theta constant and only passing in the axis. The value of theta, or the angle the pawn is rotated about the axis, is 0.1. Below are the pawn after rotating about the z-axis, x-axis, and the y-axis respectively.



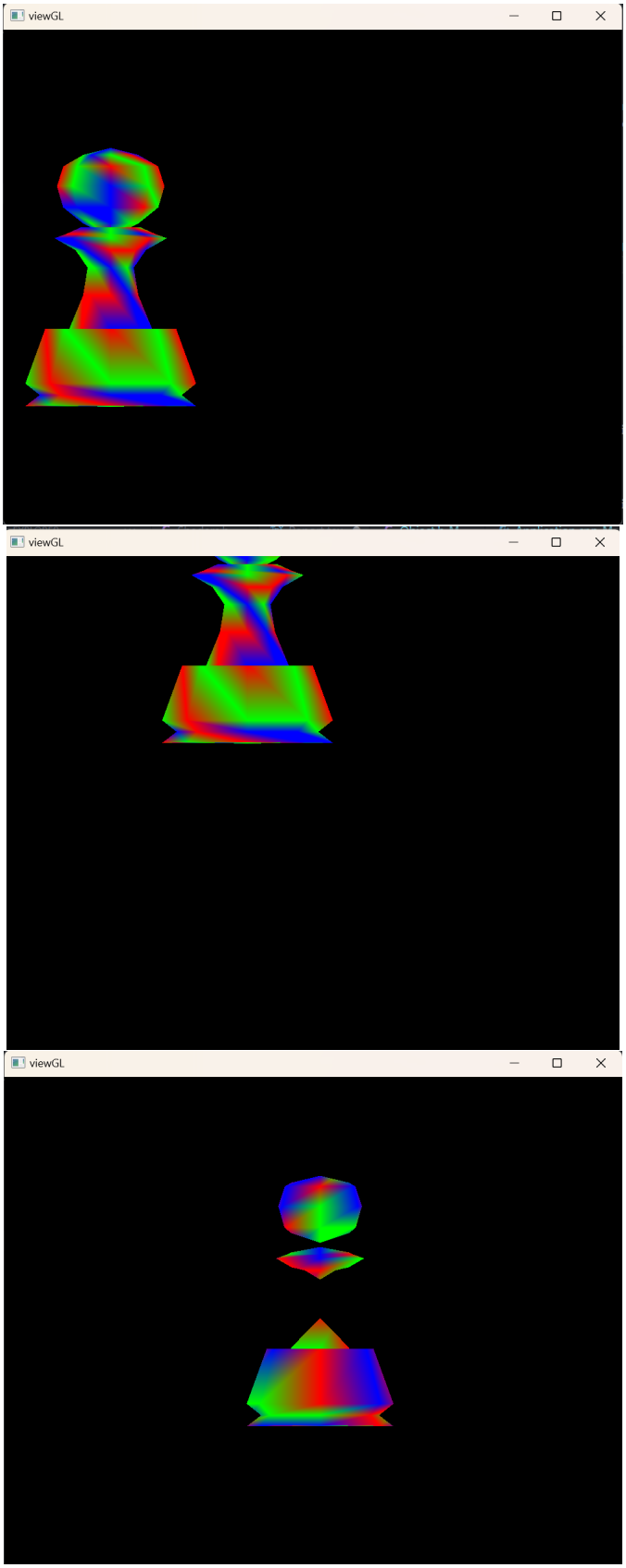


4.2 Translations

The following table shows the mapping of keys to vector that is added to the model:

Key	Translation Vector
Right	(0.001, 0.0, 0.0)
Left	(-0.001, 0.0, 0.0)
Up	(0.0, 0.001, 0.0)
Down	(0.0, -0.001, 0.0)
C	(0.0, 0.0, -0.001)
V	(0.0, 0.0, 0.001)

The follwing are the results after translating on the x-axis, y-axis, and z-axis respectively

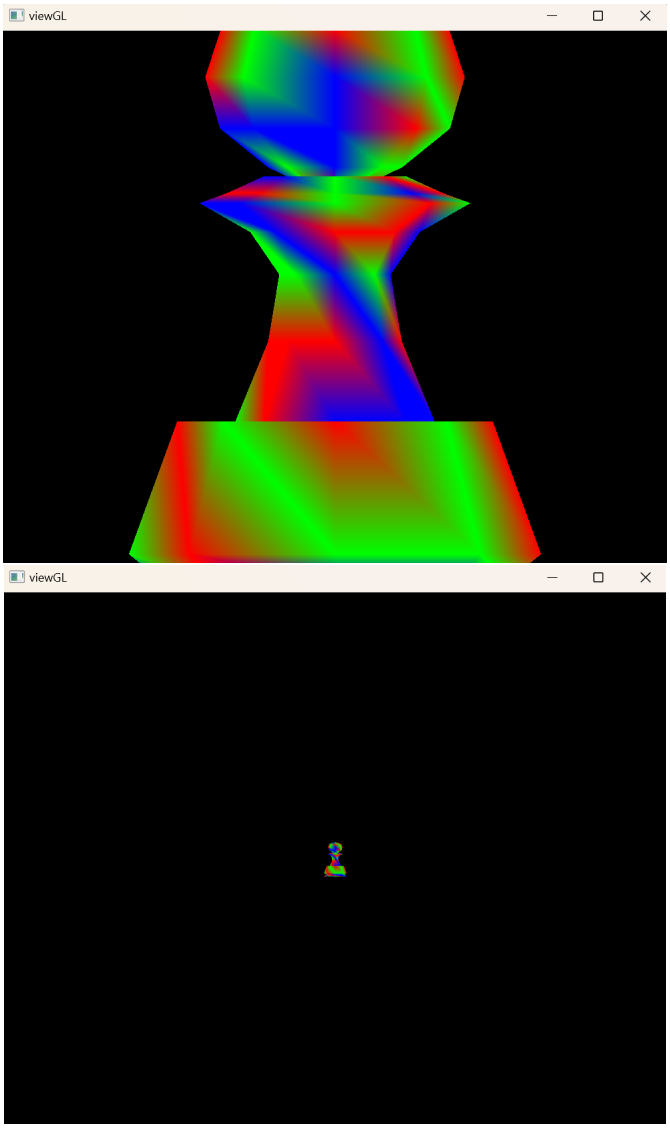

viewGL
viewGL
viewGL

4.3 Scaling

The following maps the key to the scaling vector that each component is multiplied by

Key	Translation Vector
Q	(0.995f, 0.995f, 0.995f)
E	(1.005f, 1.005f, 1.005f)

There is a small difference between 1 and both the scaling factors because otherwise the pawn would scale too quickly. Since all the components have the same scaling factor, it is an example of uniform scaling. The following are the results after scaling up and down respectively



4.4 Performance

4.4.1 GPU-Side

The first recorded performance was that of sending the transform to the GPU, the time to render was recorded by timing the length of the render while loop in microseconds. The only transformation performed is rotation. In order to get a statistically relevant value across varying random variables I took the average time to render. The following table shows the results of 5 execution trials:

Trial #	Average Render Time (μ seconds)
1	253
2	201
3	224
4	221
5	335

For the sake of comparing to CPU-sided transformations, we can take the average of these trials to get $T_{GPU} = 246.8\mu$ seconds.

4.4.2 CPU-Side

The next tested method was running the transformations on the CPU in C++ and then passing the transformed vertices straight into the shader. Following the same procedures as the GPU side, the following results are shown:

Trial #	Average Render Time (μ seconds)
1	225
2	230
3	220
4	350
5	236

Resulting in the average among these trials to be $T_{CPU} = 252.2\mu$ seconds.

4.4.3 Comparison

We then can compare the two, the GPU side is 2% faster. Typically the GPU should be faster but on the testing machine the CPU is more powerful than the GPU, even with the disparity the GPU still outperforms.