Model Transformation Assignment

Sagan Potenza

Code Structure

My code is made up of a main file for initializations and running OpenGL, the Model class to control the model, the fragment and vertex shaders (source.fs/source.vs), and 4 obj files that can be loaded (cube.obj, head.obj, shark.obj, and seesaw.obj).

The main file begins with key binds and settings that can be used to transform the models and change how models are rendered (see following sections for list of functionalities). It initializes the OpenGL through GLFW and GLEW, initializes the VAO, VBO, and EBO (if set), creates the model and adjusts the models' settings, and renders each frame when a movement is requested with the keyboard.

Key Binds

Models can be controlled in a game like interface using the following keyboard controls:

Key	Function
Escape	Exits the program.
Up/Down Arrow	Translates the model along the x-axis.
Left/Right Arrow	Translates the model along the y-axis.
Page Up/Page Down	Translates the model along the z-axis.
R/T	Rotates the model around the x-axis.
Y/U	Rotates the model around the y-axis.
I/O	Rotates the model around the z-axis.
F/G	Scales the model along the x-axis.
H/J	Scales the model along the y-axis.
K/L	Scales the model along the z-axis.
Z/X	Decreases/Increases the FOV.

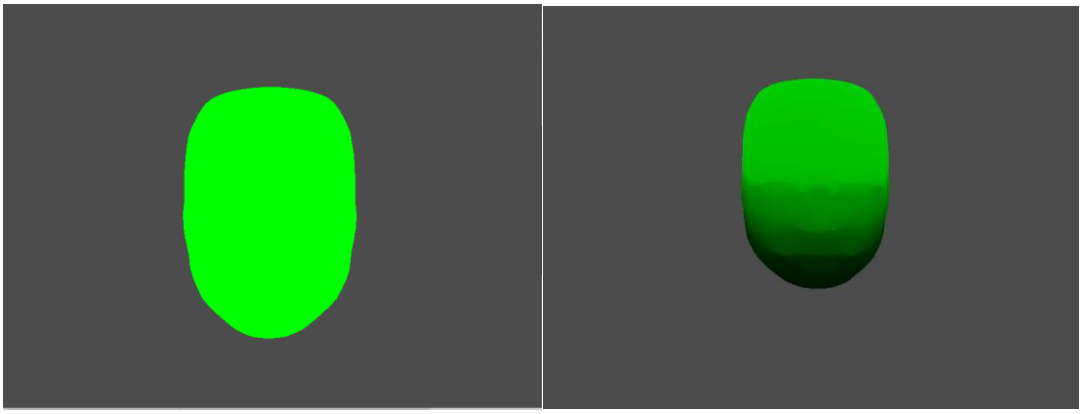
Settings

These settings are available at the start of the main function, and control the type of rendering done:

Variable	Function
int SCR_WIDTH	Screen Width.
Int SCR_HEIGHT	Screen Height.
Bool useEBO	Whether or not an Indexed Triangle Data Structure in an EBO is used (the extra credit).
Bool cpuMatrix	Whether the matrix is applied to the model on the CPU side or the GPU side.
Bool colorModifier	Whether or not Model Colors are modified.
Bool polygonMode	Whether or not Polygon Mode is active.
Bool outputPerformanceTime	Whether or not time to render each frame and current average are output.

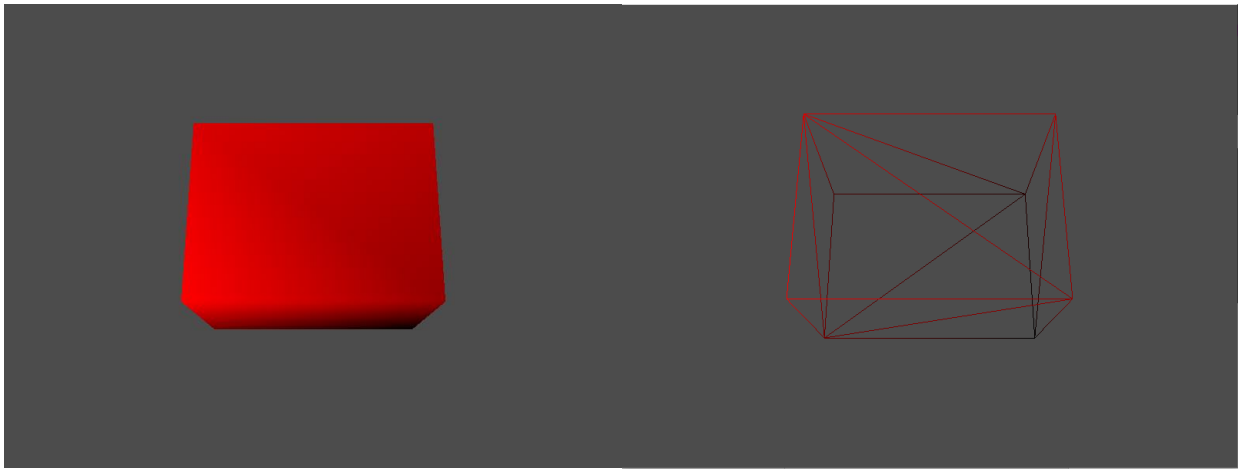
Float translationScaleStep	Amount to add/subtract when translating or scaling the model.
Float rotationStep	Number of degrees to add/subtract when rotating the model.
Float fovStep	Number of degrees to add/subtract when adjusting FOV.
Float nearClippingPlane	Near clipping distance: triangles closer than this to the camera are not rendered.
Float farClippingPlane	Far clipping distance: triangles farther than this are not rendered.
Glm::vec3 defaultColor	Color used for triangles without a material specified.
Glm::vec3 cameraTarget	Location that the camera is pointed at.
Glm::vec3 cameraPosition	Location of the camera.
glm::vec3 upVec	Up Vector of the camera.
string objFileName	File name of the .obj file to be rendered.
string vertexShaderFileName	File name of the Vertex Shader File to use.
string fragmentShaderFileName	File name of the Fragment Shader File to use.
const char* vertexMatrixUniformName	Name of the matrix uniform constant in the Vertex Shader.

Color modifier will adjust the color of each face slightly so better show the 3D geometry of models since this project does not have shading implemented (since it was not required).



Head.obj with colorModifier off vs on

Polygon Mode renders only edges of polygons, using the OpenGL setting.



Cube.obj with polygonMode off vs on

VAO

The VAO contains the VBO and EBO for the model and is setup and rendered every frame.

VBO

The VBO is generated from the generateVBOVerticesArray. It produces a vertex array that is assigned to the VBO. The Vertex array contains the x, y, z, and w of every vertex and the R, G, B, and Alpha color value, meaning each vertex takes 8 bytes in the array. The VBO is configured to read the vertex positions and colors as two separate layouts via stride settings in glBufferData. It produces a vertex for every triangle, meaning there will be repeated vertices for triangles that share vertices. It reads the color as the color of each triangle.

EBO (Extra Credit)

If useEBO is on, the generateEBOVerticesArray function will be used instead. This will produce a vertex array for the VBO containing only a list of vertices and their colors (no repeats for triangles sharing vertices) and will implement an indexed triangle data structure by producing a list of indices for each triangle for the EBO. Colors are per vertex from the colors vector.

Vertex Shader

The vertex shader is specified in source.vs and read in via the Model class. It takes in the position of each vector (layout 0) and the color of each vector (layout 1) along with the matrix (uniform) to produce a gl_Position and the vFragColor output. The gl_Position is computed by multiplying the matrix by the vertex vector (if cpuMatrix is on, this will be a Identity Matrix), and the vFragColor output is just set to the layout 1 vertex color input to pass it to the fragment shader. This results in computing the position for each vertex and passing each color to the fragment shader.

Fragment Shader

The fragment shader is specified in source.fs and read in via the Model class. It takes in the vFragColor and produces a FragColor output. All it does is pass the vFragColor input to the FragColor output.

Model Class

The model class contains 3 data structures, a Vertex data structure keeping the coordinates of each vertex, a Vertex Texture data structure which is not used (wrote in case I wanted to add texture functionality later, which was not required for this project), and a triangle data structure. The triangle data structure records the indices of its vertices, and the color of the triangle. A vector for each of these data structures is stored in the model class. Color is also stored per vertex in the colors vector for when useEBO mode is on.

The model class constructor is given an obj file name, and parses that file for every vertex and face, filling the vertex and triangle vectors. It also parses the mtl file if it exists, using the Kd diffuse constant for the color of the vertex. If a color is not found, it uses the defaultColor specified in the main file.

The model class also has the readShader() function to read in shader files, and produces character arrays containing them so the GLSL can be passed to OpenGL.

Model View Projection (MVP) Matrix

A model view projection matrix is computed in the Model class using MVP = P * V * M in getMatrix() where P is the projection matrix, V is the view matrix, and M is the model matrix.

Model Matrix

The model matrix is computed in the generateModelMatrix() function in the model class using the GLM library to create a translation matrix from a translation vector (glm::translate), rotationMatrix for each axis using an angle for that axis (glm::rotate), and a scale matrix from a scale vector (glm::scale).

The matrices are combined to create the Model matrix with M = translationMatrix * rotationMatrixZ * rotationMatrixY * rotationMatrixX * scaleMatrix. These vectors used to construct the Model matrix can all be controlled via the keyboard (see keybinds settings).

View Matrix

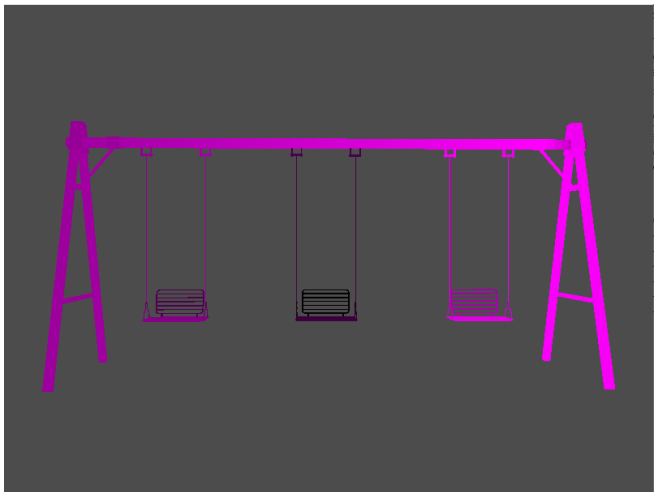
The View matrix is constructed in the generateViewMatrix() function in the Model class using the GLM library to turn the cameraPosition vector, cameraTarget vector, and upVector into the view matrix (glm::lookAt). These vectors are set in the main file settings, and can not be controlled through keyboard controls (as that was not required for the interactive interface).

Projection Matrix

The projection matrix is constructed in the generateProjectionMatrix() function in the projection class using the GLM library to turn the FOV, aspect ratio, near clipping plane, and far clipping plane into the projection matrix in perspective view (glm::perspective). The FOV can be controlled via keyboard keybinds, but the other parameters can be adjusted in the main file settings.

Performance

The cpuMatrix setting will change if the transformation matrix is applied to the vertices on the CPU or sent to the shader and applied on the GPU. This was tested on seesaw.obj, a file with 14.9k Triangles and 8.2k Vertices to test which mode had the better performance, additionally testing on useEBO mode on or off.



Seesaw.obj

The performance test can be activated via the outputPerformanceTest variable, which will time in microseconds how long each frame took to render, and output the current running average.

```
Frame 996: 4801 microseconds.
Average Time: 4323 microseconds.
Frame 997: 0 microseconds.
Average Time: 4318 microseconds.
Frame 998: 0 microseconds.
Average Time: 4313 microseconds.
Frame 999: 0 microseconds.
Average Time: 4308 microseconds.
Frame 1000: 997 microseconds.
Average Time: 4304 microseconds.
```

Example Output of Performance

This was done for each setting combination for 1000 frames:

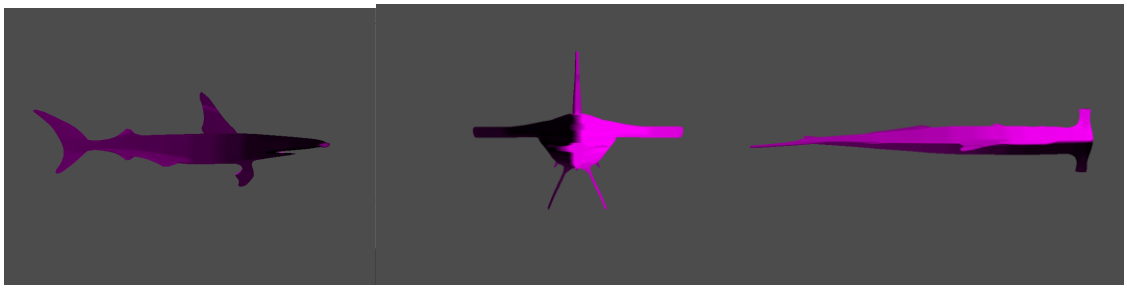
(cpuMatrix, useEBO)	Average Time Per Frame After 1000 Frames (µs)
(false, true)	4252
(false, false)	4253
(true, true)	4370
(true, false)	11350

This resulted in cpuMatrix off with useEBO on having the fastest time of 4252 µs which makes sense as it's applying the matrix on the GPU and using the more efficient indexed triangle data structure with the EBO. With cpuMatrix off, but useEBO also off, it didn't increase the time of 4253 µs, implying that the GPU was the cause of the fast performance. When cpuMatrix was turned on so that the matrix was applied on the CPU, this resulted in a slightly longer time of 4370 µs with useEBO on, but a drastically longer time of 11350 µs with useEBO off, meaning that useEBO helps mitigate the slower

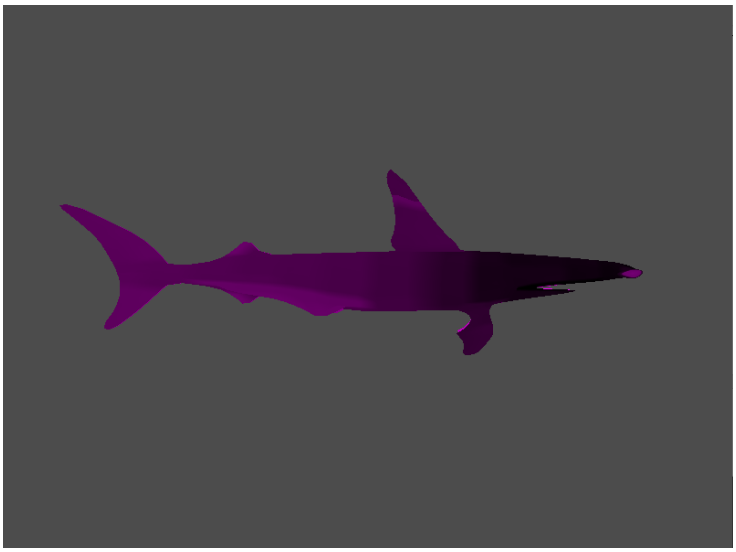
time from running the matrix on the CPU. Overall, it is clear that it's much faster to send the transformation matrix to the GPU rather than transform the coordinates on the CPU, and that using an indexed triangle data structure can decrease the number of vertices, allowing for better performance.

Transformation Demos

The ModelViewProjection matrix is constructed via GLM libraries and can be controlled with the Keyboard to transform the model. The View matrix is controlled via settings at the beginning of the main file, but the model/view matrices can be controlled via the keyboard to transform the model. A hammerhead shark model Shark.obj was used to demonstrate these capabilities:



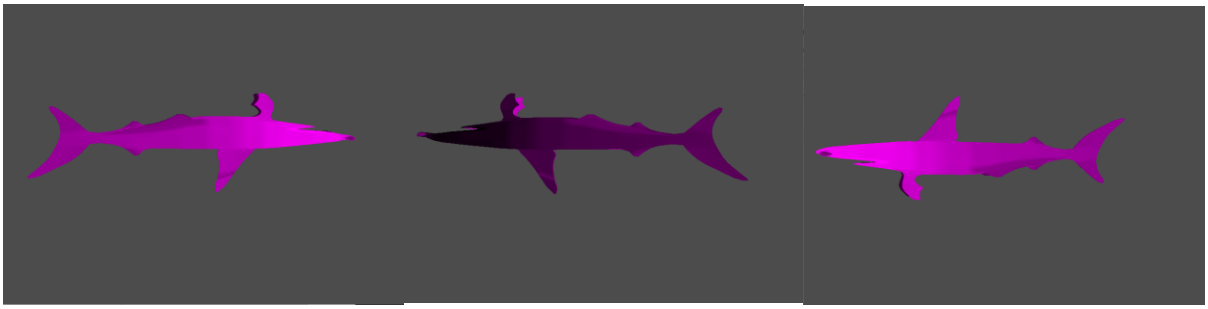
Shark.obj from various angles.



Default Position Used for the Following Demos

Rotation

Rotation is controlled in the Model Matrix.



Rotated 180° on X-Axis vs Y-Axis vs Z-Axis

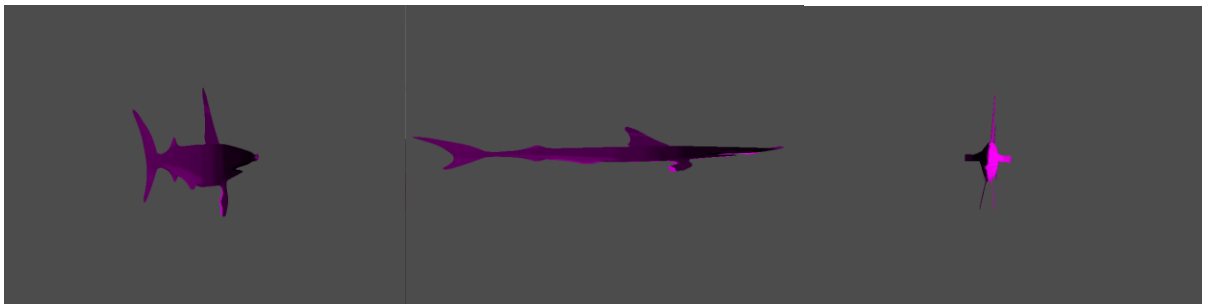
Translation

Translation is controlled in the Model Matrix.

Translated about X-Axis vs Y-Axis vs Z-Axis

Scale

Scale is controlled in the Model Matrix.

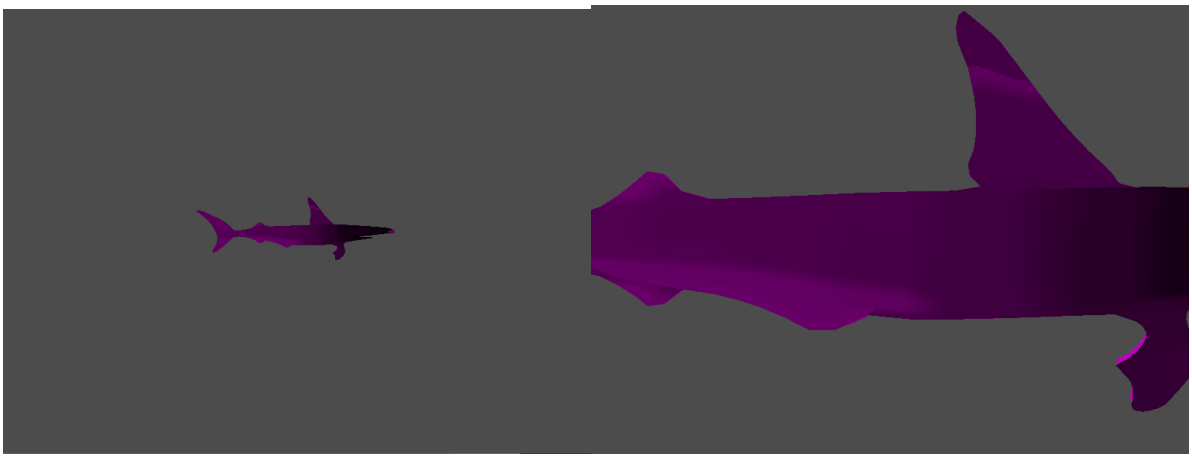


Scaled Down About X-Axis vs Y-Axis vs Z-Axis (Z-Axis is rotated to better see change)

Scaled Up About X-Axis vs Y-Axis vs Z-Axis (Z-Axis rotated to better see change)

FOV

FOV is controlled in the Projection matrix.



Increased FOV vs Decreased FOV

Video Demo

A demo for this project and the transformations is available at https://www.youtube.com/watch?v=0Z2WLjYUkPk.

Sources

This project used the GLFW, GLEW, and GLM Libraries. Other than this, no outside libraries were used, and no code was copied or repurposed from any sources, except for repurposing the initial main file from the professor. The following sources were referenced for technical information:

- Main file has portions of code from the initial main file provided by the professor.
- https://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices/ was referenced to learn about constructing Model View Projection matrices with GLM and OpenGL.
- https://learnopengl.com/Getting-started/Hello-Triangle was referenced to learn about EBOs for the extra credit.

- https://sketchfab.com/3d-models/recreational-place-play-children-seesaw-159f3bf2bc1f472da8a94d0dc5c8aa3f is where seesaw.obj was found.
- https://www.dropbox.com/s/ffejwevoylh4kjz/data.tgz?e=1&dl=0 is where shark.obj, head.obj, head.mtl, cube.obj, and cube.mtl were found.