**7-2 Final Project: Reflection**

Alex Baires

Southern New Hampshire University

CS-330-J8767 Computer Graphics and Visualization

Professor Eugenio Rodriguez

February 25th, 2024

Working on this project was a labor of love. It enabled me to combine my ability to troubleshoot 3D video games, my proficiency in C++, and my creative side. My sister is an artist, and she was floored with how one can use matrix and vector operations to create 3D effects (like lighting and perspective), similar to the processes she learned in her formal training.

My choices when creating this project could be divided into design and development. The design portion consisted of determining suitable objects to render in OpenGL. I intended to choose complex yet straightforward shapes that were visually interesting while gradually increasing their rendering difficulty. The selenite crystal was a stepping stone, allowing me to combine the two shapes I created from the first 3D model tutorial and assignment. Though simple, its reflective and rough qualities enabled me to further refine the final project's lighting effects and texture filtering. I applied the lessons learned from creating this model to the rest of my objects. The traffic cone was a natural progression because it had a conical top with a rectangular base. The solar system sphere with a pedestal serves as the centerpiece of my scene. I created a custom tapered polygon mesh for the bottom. Additionally, I used an algorithm that produces a procedurally generated sphere found in the book *Computer Graphics Programming in OpenGL with C++* by V. Scott Gordon and John Clevenger. I used different material properties in a struct inside of my fragment shader for each model. The resulting effect of the sphere’s silver material reflecting each light source was aesthetically pleasing, which made it challenging to texture. Instead of texturing, I decided to recreate the globe with its small planetary spheres by using GL\_BLEND and alpha values to make the parent, encompassing sphere transparent within its own fragment and vertex shaders. The challenge was the order in which the objects would be rendered; the sphere would have to be rendered last to see the rest of the objects (both the planets and the other models) through its surface. To prevent artifacts caused from rendering the sphere’s triangles out of order, I also used a two-pass blending solution, where I first enabled front-face culling to generate the rear of the sphere and then enabled back-face culling to render the front. In contrast, if I had used a texture effect, I would not have accomplished the miniaturized 3D scale of the solar system within the sphere.

The model of the Imperial TIE Fighter from *Star Wars* is a culmination of the work I completed throughout the class and my development decisions. It is a hierarchal model consisting of about 29 primitive shapes, and this model could only be accomplished because of my decision to utilize a matrix stack data structure during development. The view matrix is pushed onto the stack, and each model has a parent shape (in the TIE Fighter’s case, the spherical cockpit) with translation, rotation, and scale matrices added to the stack. The additional shapes (like the wings) are child shapes of the cockpit and inherit its positioning. Adjustments to the child shapes are relative to the parent shape, making it easier to move them within the model’s local space. A drawback to this approach is its complexity when multiple matrices are in the stack, but I had a pen and paper handy to keep track. Another option would have been to manually multiply each child model matrix by the parent model-view matrix, but this would have been complicated and time-consuming.

The scene was designed to be navigated with keyboard and mouse input devices using the Camera.h class and functions from *LearnOpenGL.* In my utils.cpp file, the processInput() function manages the GLFW keyboard input commands. The user can press the W, S, A, and D keys to move in the forward, backward, left, and right directions. The Q and E keys are used to pan the camera up and down. I also added an IF/ELSE statement for the F key which changes the rendering polygon mode to wireframe instead of fill (the default) on command.

A screen shot of a computer code

Description automatically generated

Additionally, the activateOrtho() function also allows the user to trigger an orthographic projection of the scene that removes perspective and can be toggled by pressing the P key.

A screen shot of a computer code

Description automatically generated

To look around the scene, I first disabled the mouse cursor and kept it within the center of the window using glfwSetInputMode(). The mousePositionCallback()function gets the mouse’s current position and calculates the mouse’s x and y offset values. It then pushes these values to the camera class’s ProcessMouseMovement() function, where it calculates the direction, multiplies it by a sensitivity value, and updates the camera vectors in real-time.

This project was designed with modularity in mind. Each vertex and fragment shader has been removed from the main.cpp file and put inside its own .glsl file. The custom function createShaderProgram()takes both file paths and uses the custom readShaderSoruce() function to parse through each line of the .glsl file and append a new line character per line using file streams. This function also does error checking and compiles and attaches the shaders. Having both shaders in separate files allows me to reuse them in future OpenGL programs, provided that I also use the custom loading functions. Fragment shaders are powerful at generating animation and effects by utilizing the GPU’s shader processing power. I experimented with using multiple modular shaders when adding transparency to my solar system globe without impacting the other models. A future program could have shaders that add fog effects or even explosions. All of my meshes are in a separate meshes.cpp file that can be used in other programs by including the header and .cpp files and then using the createMeshes() function to have them ready to use. For my complex objects, I encapsulated them within their own draw() functions. For example, drawTIEFighter() can repeatedly draw a TIE Fighter object with a vec3 position as an argument, perfect for games where multiple enemies can respawn. However, I could have added a rotational argument to the function so the fighter could be rotated differently in each draw() call. If I had more time, I would have further encapsulated my lighting code to utilize inheritance to create a Light class that can generate other types of light sources.