Project Two Design Decisions

CS-330 23EW4

Jared Smith

4/15/23

When I originally wrote my project proposal for this class, I didn’t really realize what I was getting myself into. I had originally chosen a crowded scene with lots of objects, and when that was rejected, I swapped over to a scene with 5 simple objects representing all the simple shapes except pyramid. During the development of the project, I ran out of time to write a algorithm for generating a torus mesh, so I ended up going with the four remaining simple objects in my scene, represented by a plane, two cubes, a few cylinders and a sphere. I chose the objects because I thought they would be interesting to model and texture.

Generating the cube and plane was fairly simple, I wrote the vertex information by hand after modeling the cubes in Microsoft paint 3d, operating from -1 to 1 on each axis. For the cylinders and sphere, modeling each vertex by hand was not a viable solution, so I adapted a few algorithms from <http://www.songho.ca/opengl/index.html>, which generate vertex and index information for each shape. These return a generic vertex and index array, which can be scaled and textured for use. With a little extra effort, each function could also be modularized to generate vertices for other similar symmetrical objects, in the case of spheres algorithm, things like diamonds, and octahedrons. In the case of the cylinder algorithm any multi-sided prism could be generated. These could be further modularized to make asymmetrical shapes, but I didn’t find that necessary for this project so I didn’t make time to do them.

The next step in generating the scene was to create objects that could be accessed from the code and by openGL easily, so I defined a GLMesh struct that contained vital information, like the VBO and VAO ID’s, number of vertices, size of the arrays, etc. I then created a UCreateMesh function that took the mesh and array information and bound them to openGL’s VAO and VBOs, and retrieved those corresponding IDs. Now that we have the meshes defined for openGL, we can go about loading and buffering our textures, then defining our render loop.

The render loop is nothing too special, it keeps frame timing for our input reading functions, enables depth test and clears the gl buffers so a fresh frame can be drawn, then starts activating textures and drawing the corresponding objects. I used a single URenderTriangles function to draw each object to the screen. This function determines the perspective type by checking a global Boolean variable, and creates the appropriate view and projection matrix. A model matrix was predefined for each object, and passed to the function and those three matrices are bound to the uniforms in our shader program. We also passed some lighting information here in this function so the fragment shader can calculate the effect of light on each vertex, and bound up the texture information to same. There is then a simple check that determines if the mesh object being drawn has index data, and calls the appropriate draw function. glDrawArrays is called for meshes with no index data, and glDrawElements for meshes with index data. The reason for this little difference is mostly due to my learning better ways of modularizing the drawing process as the project was developed, but the solution is sufficient for the project’s purpose, so I didn’t refactor it. The reason we needed a way to draw by index was to avoid an unnecessary amount of vertex information for complex shapes (the cylinder and sphere, namely). To make a round sphere, we used 22,568 unique vertexes requiring that we draw to 16,020 indices. If I had instead inserted duplicate vertices into the array, they would have taken up over 8 times the amount of memory (not including the disparity in storing integers vs floats).

Lastly in the program flow, we activate our light program shader and poll events, and in the case the window was closed (or the esc key was pressed) we destroy our meshes and programs to clean up our memory and allow the program to exit.

The scene is navigated using WASD controls to move along the Y and X axis, relative to the camera direction. The camera is controlled by the mouse, using pre-built glfw functions. You can also navigate along the Z axis using Q and E. You can accelerate and decelerate your movement speed using the mouse wheel, and can change to an orthographic perspective by pressing P.

As far as modularity, having the mesh creation dependent solely on having a mesh struct and an array of vertex data makes creating new objects fairly easy, provided you can model or generate an array of vertex data. If your vertices are ordered to be drawn sequentially, with duplicates that can form a whole object made of triangles, all you have to do is define your texture file path, load the texture, bind it and send them off to the render function. If you have index data, its as simple as defining the number of vertices for your mesh object and sending them to the render function. Obviously some of the functionality of my program could be abstracted, but for the purposes of this assignment I believe there is more than sufficient modularity.

**Reference**:

Ahn, S. H. (n.d.). *OpenGL*. http://www.songho.ca/opengl/index.html