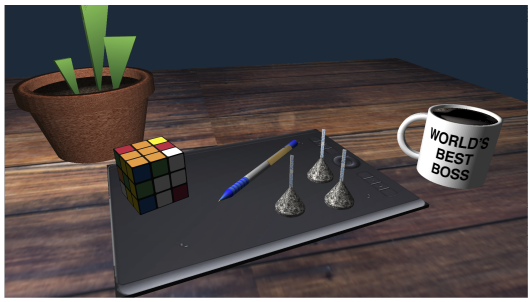
Lawrence Artl
CS-330 22EW5
Module 7: Final Project Reflection
June 8, 2022

Final Project Reflection

With the final project submitted, I can't help but look back at the journey that I just embarked on. Over the course of six weeks our class went from developing simple text-based console applications to rendering fully immersive 3-dimensional scenes with light. That's a pretty amazing feat, and one that helps to showcase just how amazing computers are. In this reflection paper, I would like to discuss some of the elements of my application that made this journey possible, with a focus on the algorithms and methods that I developed myself. While my final project does not perfectly match my original concept, this was done by design as I learned new techniques and found more appealing textures to use (some of which were created from scratch in Adobe Illustrator and Photoshop). The overall spirit of the scene remains the same, though, and I hope it can be appreciated in spite of the differences.

Original

As Rendered



From the very beginning I wanted to make the program modular so I could reuse portions of it for future builds. This required separating the algorithms for shape building from the methods used to render the scene. Additionally, I felt it would make the application more user-friendly to consolidate all scene-creating items into their own class so that the user can focus on just adding shapes and texturing them, without having to scroll through hundreds of lines of code. In order to do this I refactored the GLMesh struct to take in all the properties of the shapes, then the mesh object gets passed to the ShapeBuilder class to be constructed. Once the shape is constructed and textured (and scaled and transformed in the world) the mesh object is added to a vector called "scene". The vector is looped through during the Render method, and each shape is extracted and drawn on screen.

```
// PEN BODY
    GLMesh cyl_gMesh01;
    cyl_gMesh01.p = {
        1.0f, 1.0f, 1.0f, 1.0f,
        0.25f, 1.0f, 0.25f,
        -90.0f, 1.0f, 0.0f, 0.0f,
        0.0f, 0.0f, 1.0f, 0.0f,
        0.0f, 0.0f, 0.0f, 1.0f,
        0.0f, 0.10f, 3.0f,
        1.0f, 1.0f
    };
    cyl_gMesh01.height = 4.0f;
    cyl_gMesh01.radius = 0.5f;
    cyl_gMesh01.number_of_sides = 128.0f;
    cyl_gMesh01.texFilename = "textures\\pen_body.png";
    ShapeBuilder::UBuildCylinder(&cyl_gMesh01);
    scene.push_back(cyl_gMesh01);
```

This set-up helped to save code, and allowed me to reuse algorithms easily. For instance, when creating a new cylinder shape I just needed to make a new mesh object with the required properties (size, rotation, texture, etc). The entire scene can be built by the user in one class (“SceneBuilder”) without having to make changes to any other class or file.

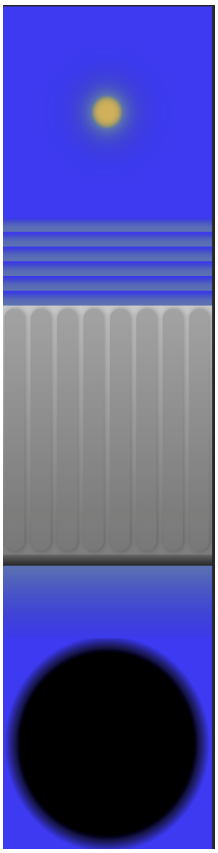
The navigation component of the application contains a variety of functions that the user can use to manipulate what is shown on-screen. The standard motion of the camera uses the WASD keys for basic movement, Q and E for up and down movement, and the mouse to change the view of the camera. These keys / motions use the camera.h predefined motions to adjust the camera’s position within the scene.

Additionally, to add some “testing” functionality to the application the user can use the opposite end of the keyboard to adjust the spotlight’s position within the scene. The keys IJKL move the light around within the x and z planes, while U and O move it up and down. The light can revolve around the scene with the left and right alt keys, while the directional light from above can be turned on and off with the left and right brackets. Finally, the view perspective can change with V and B keys, and the shapes can be shown in wire-frame mode with the left and right arrows.

Since the camera and light positions are determined by vectors for location, modifying these vectors ultimately modifies the location of the object. As the user presses each key, the value at the specified location within the vector is adjusted, either by the camera.h file or within the main source code. The object then appears to move. What’s most interesting about this concept is the fact that when the user looks at what is rendered on screen they are not looking at a 3-dimensional scene. Rather, the libraries of OpenGl take the coordinates and properties of what would be a 3 dimensional object and render it within a 2-dimensional space, their perspective adjusted to give the appearance of 3-dimensions. It is the same concept as when an artist paints a picture using light and shadow to give the illusion of the 3-dimensions on a 2-dimensional canvas. The biggest difference here is that the scene is re-rendered with new properties as per the calculations of the libraries every time the camera position or view is adjusted.

The most intriguing functions in my application are the shape-building methods in the “ShapeBuilder” class. Each method was written from scratch by myself, with little outside help save for a calculator and trial / error. The cylinder function, for example on the right, makes use of the mathematics needed to find a point along the edge of a circle based on the radius of said circle. By calculating the cosine and sine of 2π times the section of the circle being drawn (a number counting upwards from 1 and ending at the maximum number of sections of the circle), multiplied by the radius and the starting coordinate, the necessary point in 2-dimensional space can be found. Using this calculation the method draws a fan of triangles with all starting points at the center origin and all other points radiating outwards. This same process is repeated ‘h’ height away, followed by another process to connect the two circles with a column of triangles.

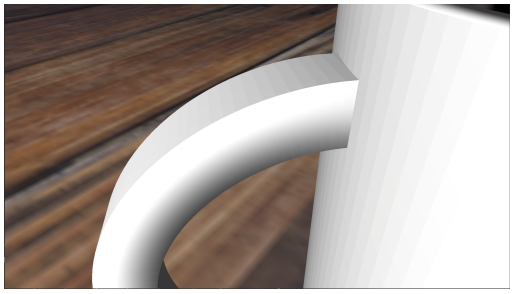
```
0.5f, 0.0f, 0.5f, 0.0f, -1.0f, 0.0f, 1.0f, 0.5f, 0.125f });        //
0.5f + r * cos(_Xx: i * sectorStep) ,        // x
0.0f ,                                       // y
0.5f + r * sin(_Xx: i * sectorStep) ,        // z
c[0], -1.0f, c[2], 1.0f,                        // color data r g b a
0.5f + (0.5f * cos(_Xx: (i)*sectorStep)) ,      // texture x; adding th
(0.125f + (0.125f * sin(_Xx: (i)*sectorStep))) });       // texture y


0.5f + r * cos(_Xx: (i + 1) * sectorStep) ,
0.0f ,
0.5f + r * sin(_Xx: (i + 1) * sectorStep) ,
c[0], -1.0f, c[2], 1.0f,                        // color data r g b a
0.5f + (0.5f * cos(_Xx: (i + 1) * sectorStep)) ,
(0.125f + (0.125f * sin(_Xx: (i + 1) * sectorStep))) });
```



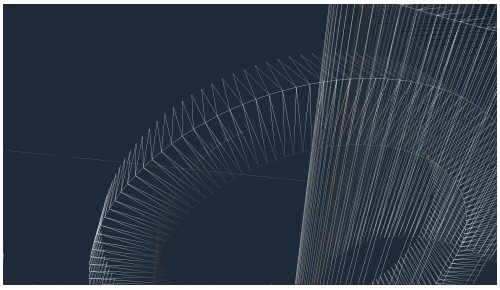
Calculating the texture coordinates for complex shapes like cylinders (example on the left) and cones proved to be challenging at first, but an elegant solution was devised. The end goal was to have the shape, a cylinder for instance, have one texture for the top, another for the bottom, and a third to wrap around the central column. Since texture coordinates live within a 0 to 1 range in both the x and y axes, special care had to be taken to ensure that the correct portion of a texture file was read. As mentioned above, using cosine and sine to determine the coordinate point on the circle was helpful, while keeping in mind the new central position of the calculated circle. The texture files for cylinders are tall and thin, with the top 25% for the top of the cylinder, the bottom 25% for the bottom of the cylinder, and the central 50% for the column of the cylinder. Considering these locations, the top of a cylinder gets it’s texture from calculating the coordinates of a circle whose origin is 0.5 in the x-axis, but 0.125 in the y-axis.

The hollow cylinder (used to build the coffee mug and handle) uses the above mentioned algorithm but modified to create two cylinders, one inside the other. They are both connected by a truncated triangle fan at the top and bottom; that is, the fans are built with a “hole” in the center. This was achieved by instead drawing the fan as a “strip” of triangles, one that meets at both ends as a circle (see below).. Had there been more time, I would have used a similar algorithm to construct a sphere out of strips but alas, the course is only eight weeks.

Solid Filled Hollow Cylinder



Wire-Frame Hollow Cylinder



With the project now complete and polished, I would like to reflect on the purpose of the assignments of this course. While one goal was to learn OpenGl and expose us as students to the possibilities the libraries hold, I feel that the larger goal was to allow us to practice with designing our own algorithms to solve problems. The problems in this case were how to render 3-dimensional shapes, create movement within the scene, etc, but the art of problem-solving as a whole was the main focus. The various assignments and subsequent applications gave me a lot of opportunities to encounter new issues and thus flex my problem-solving “muscles” to create solutions to those issues. Additionally, the end result was something tangible, a piece of software that other people (laypersons, mainly) can appreciate and enjoy. Most don’t understand the work that goes into creating a program, much less when that program is a simple console application. This was a nice way to showcase to others what computers (and their programmers) are capable of.