CAP 5705, Project 2: OpenGL Warmup

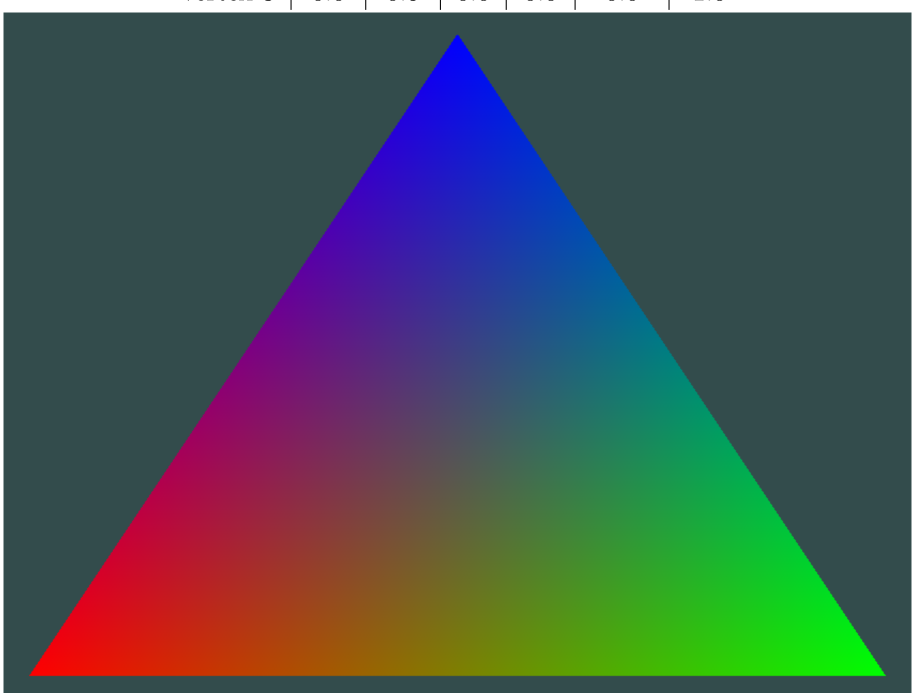
Davis Arthur, davisarthur@ufl.edu

October 8, 2021

1 Color

In the `helloTriangle.cpp` file, I altered the Vertex Buffer Objects and Vertex Array Objects to include a color for each vertex. To do so, I represented each vertex with 6 float values. The first three float values of a vertex indicate the vertex's position. The second three float values of a vertex indicate the vertex's RGB color. Each element within the color portion of the vertex data must be within the range [0, 1]. Below I have taken a screenshot of the output of my modified `helloTriangle.cpp` program with the following vertex configuration:

	x	y	z	red	green	blue
vertex 1	-0.5	-0.5	0.0	1.0	0.0	0.0
vertex 2	0.5	-0.5	0.0	0.0	1.0	0.0
vertex 3	0.0	0.5	0.0	0.0	0.0	1.0



2 Reading Vertex and Fragment Shaders from a File

In each of my rendering programs except `helloTriangle.cpp`, the vertex shader and fragment shader are read and loaded into the OpenGL program using the *readFile* function defined in my `helperFunction.cpp` file. Below I have included screenshots of the two sections of my source code that provide this functionality. The left image is from my `helperFunction.cpp` file. The right image is from my `cube.cpp` file.

```
string readFile(string fileName) {
    string output = "";
    string line;
    ifstream myfile (fileName);
    if (myfile.is_open()) {
        while (getline(myfile, line)) {
            output += line + "\n";
        }
        myfile.close();
    }

    else cout << "Unable to open file";

    return output;
}
```

```
    // read vertex shader
    string vertexShaderSourceString = readFile("source.vs");
    char* vertexShaderSource = &vertexShaderSourceString[0];

    // read fragment shader
    string fragmentShaderSourceString = readFile("source.fs");
    char* fragmentShaderSource = &fragmentShaderSourceString[0];
```

I have created several vertex and fragment shader files that are discussed in detail in Section 4. Each of these can be loaded using *readFile*.

3 Reading Meshes from a File

First, I created a struct *vertexData* to contain the data of each vertex. The *vertexData* struct contains an element of type `glm::vec3` named *pos* indicating the position of the vertex before any transformation is applied. Next, I created a struct *Triangle* that contains three *vertexData* elements: *vertex1*, *vertex2*, and *vertex3*. These three elements indicate the vertices of the triangle in counter clockwise order. My separate triangles data structure is implemented using a vector of type *Triangle*.

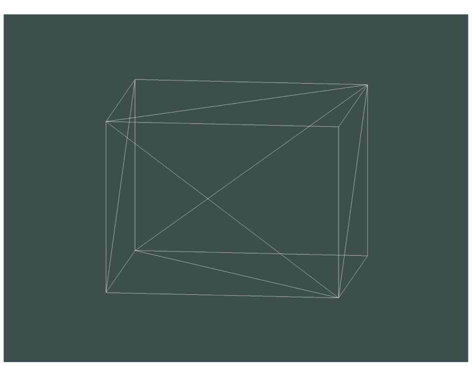
Next I created a function *readVertexData* (in the `helperFunctions.cpp` file) that reads a mesh from a `.obj` file and returns the corresponding separate triangles data structure. This function first reads each vertex in the `.obj` file. Next, the function reads the vertex indices of each face. Let the vertex indices of a particular face be denoted by a vector $\boldsymbol{f}$ of length n. Each face is an n-polygon that will need to be constructed from $n-2$ triangles. The ith triangle ($1 \leq i \leq n-2$) is constructed using the following vertex indices f_1, f_{i+1}, f_{i+2}. Each triangle is added to the separate triangles data structure immediately after it is constructed. After all $n-2$ triangles of a face are constructed, the *readVertexData* function moves on to the next face. When all faces have been constructed, the separate triangles data structure is returned.

My `cube.cpp` program reads and renders the model within the `cube.obj` file. I rendered the mesh with and without wireframe mode enabled. The `source.vs` vertex shader is used, which transforms each vertex so all vertices are visible in the image. The transformation matrix M that is passed to the vertex shader is the product of a camera matrix M_{cam} and an orthographic projection matrix M_{ortho}.

$$M = M_{\text{ortho}} M_{\text{cam}}$$

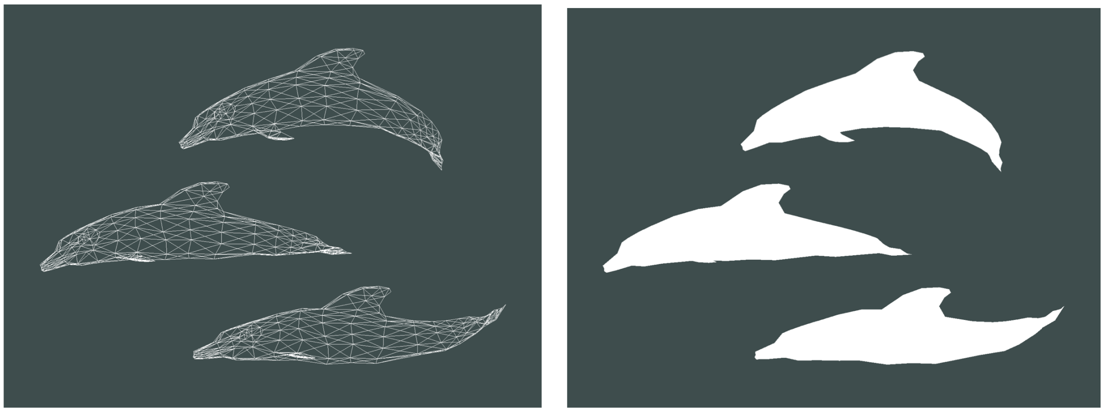
The orthographic projection matrix and camera matrix were created using the parameters given in the table below. For this rendering, I used my `source.fs` fragment shader, which colors each vertex white.

orthographic matrix						camera matrix		
left	right	bottom	top	far	near	eye	lookAt	up
-2.0	2.0	-2.0	2.0	-10.0	10.0	(0.5, 1.0, 4.0)	(0.0, 0.0, 0.0)	(0.0, 1.0, 0.0)



My `dolphins.cpp` program reads from the `dolphins.obj` file. I rendered the triangles with and without wireframe mode enabled. The orthographic projection matrix and a camera matrix for this model are defined in the table below. For this rendering, I used my `source.fs` fragment shader again, which colors each vertex white.

orthographic matrix						camera matrix		
left	right	bottom	top	far	near	eye	lookAt	up
-500.0	500.0	-500.0	500.0	-100.0	100.0	(0.0, 0.0, 10.0)	(0.0, 0.0, 0.0)	(0.0, 1.0, 0.0)



My `flowers.cpp` program reads from the `flowers.obj` file. When reading this file, I used my `colorCoords.vs` vertex shader and my `colorCoords.fs` fragment shader (more details in Section 4.5). This vertex shader applies the same position transformation as the `source.vs` vertex shader. The orthographic projection matrix and camera matrix used are defined in the table below.

orthographic matrix						camera matrix		
left	right	bottom	top	far	near	eye	lookAt	up
-20.0	20.0	-20.0	20.0	-10.0	10.0	(0.0, 1.0, 5.0)	(0.0, 0.0, 0.0)	(0.0, 1.0, 0.0)

4 Modifying Shader Files

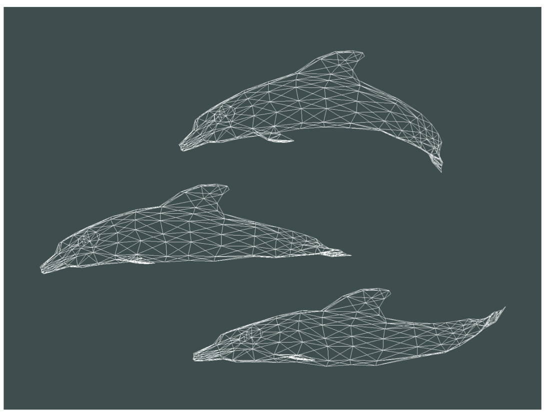
I made 5 vertex and fragment shader combinations that can be loaded within my `dolphins.cpp` file. The details and effects of each are explained in the following subsections.

Mode	Code	Vertex Shader	Fragment Shader
Default	0	`source.vs`	`source.fs`
Custom Color	1	`source.vs`	`customColor.fs`
Swap Coordinates	2	`switchCoords.vs`	`source.fs`
Scale Coordinates	3	`scale.vs`	`source.fs`
Color Coordinates	4	`colorCoords.vs`	`colorCoords.fs`

When running my `dolphins.cpp` program, the terminal will first prompt the user to enter a shader code indicating which shader combination they would like to use. Next, the program will proceed to ask the user whether the mesh should be rendered with or without wireframe mode enabled. After the user answers all prompts, the mesh is rendered.

4.1 Default

The default vertex shader simply applies the orthographic projection and camera transformation (described in Section 3) to each vertex to calculate its final position in the canonical view volume. The default fragment shader colors each vertex white.




4.2 Custom Color

The custom color shader mode will prompt the user to enter 3 numbers between 0 and 1. These three numbers are used to configure the RGB color of each vertex in the fragment shader. Below I have displayed the output for 3 possible user inputs.

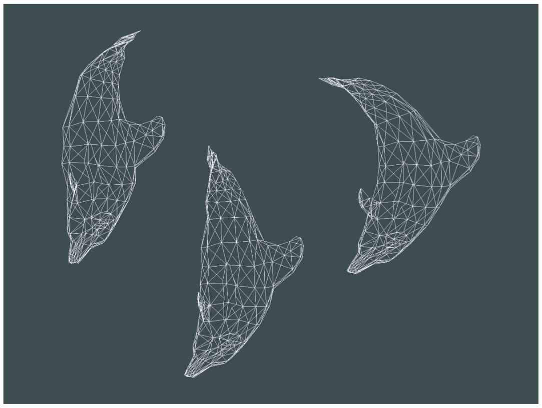
$(0.5, 1.0, 0.0)$

$(0.5, 0.0, 1.0)$

$(0.0, 0.5, 1.0)$

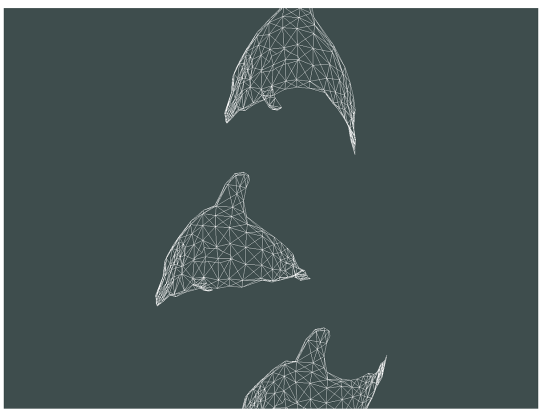
4.3 Swap Coordinates

The swap coordinates shader mode switches the x and y position of each vertex within the vertex shader. The default fragment shader is still used, so the mesh is colored white. Because the screen width is larger than the screen height, the dolphins model is distorted after this transformation. The body width is increased, and the body length is decreased.




4.4 Scale Coordinates

The scale coordinates shader mode scales the x position by a factor of 0.5 and scales the y position by a factor of 1.5 within the vertex shader. This has the effect of decreasing the length of the dolphin and increasing the width of the dolphin. Additionally, the top of the uppermost dolphin, and the bottom of the lowermost dolphin is clipped after scaling. The default fragment shader is still used, so the mesh is colored white.



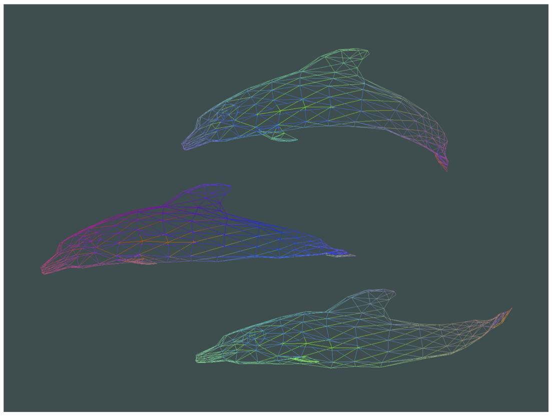
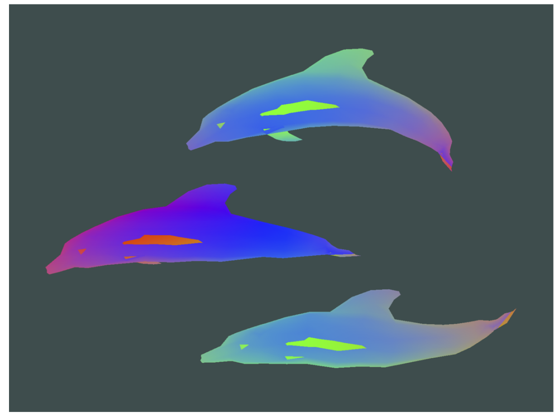

4.5 Color Coordinates

The color coordinates shader mode is slightly more complex than the other modes. First, the vertex shader outputs a vec3 named color. Let $\boldsymbol{p} = (x, y, z)$ denote the postition of a vertex after the camera and projection transformation has been applied. The output color vector is defined below:

$$\text{color} = \frac{(|x|, |y|, |z|)}{\sqrt{x^2 + y^2 + z^2}}$$

The color vector is an input to the fragment shader. The fragment shader uses it as the RGB color output of the corresponding fragment. This shader has the following effects:

1. Vertices on the far right or far left side of the screen are colored red.
2. Vertices on the top or bottom of the screen are color green.
3. Vertices near or far from the camera are colored blue.

5 Notes

- The orthographic matrix and camera matrix are generated using the `glm` library. This library can be installed on MacOS using the command `brew install glm`.
- The *split* method in my `helperFunctions.cpp` file is based loosely on a tutorial from the following website: `https://www.geeksforgeeks.org/how-to-split-a-string-in-cc-python-and-java/`
- The *readFile* method in `helperFunctions.cpp` file is based off a tutorial from the following website: `https://www.cplusplus.com/doc/tutorial/files/`