CS 4600 Homework 3

YouTube: https://www.youtube.com/watch?v=52JL518yXfY

Part 1: Compute Normals

In order to compute the normal vectors, I first found the indices of each index of g_meshIndices. These indices are the polygons and would be used to calculate the vertices. In order to calculate the vertexes, I pass the indices into a function the simply subtracts the two indices. Once the vertices are found, these are passed into a cross product function that returns the cross product. Both the cross-product method and the vertex method were created by me as helper methods simply because I didn't realize until right before the assignment submission that there were already methods made to do this. I also wanted to implement them myself to fully gain an understanding of what was going on. Once I had the cross-products, these were normalized by dividing by the length. I then added the normalized values back to gl_meshNormals, and running a for loop over the gl_meshNormals for each index i, i+1, and i+2.

```
// had to change this to indices, wasnt filling in the middle section of the teapot
for (int v = 0; v < g_meshIndices.size() / 3; ++v)
{

    //replace thise
    //g_meshNormals[3 * v + 2] = 1.0;
    int indice1 = g_meshIndices[3 * v];
    int indice2 = g_meshIndices[3 * v + 1];
    int indice3 = g_meshIndices[3 * v + 2];

    //get vertices using indices
    float ux = getVertex(g_meshVertices[3 * indice1], g_meshVertices[3 * indice2]);
    float uy = getVertex(g_meshVertices[3 * indice1 + 1], g_meshVertices[3 * indice2 + 1]);
    float uz = getVertex(g_meshVertices[3 * indice1 + 2], g_meshVertices[3 * indice2 + 2]);

    float vx = getVertex(g_meshVertices[3 * indice1], g_meshVertices[3 * indice3]);
    float vy = getVertex(g_meshVertices[3 * indice1 + 1], g_meshVertices[3 * indice3 + 1]);
    float vz = getVertex(g_meshVertices[3 * indice1 + 2], g_meshVertices[3 * indice3 + 2]);

    // get cross product using vertices
    float crossX = getCrossProduct(uy, vz, uz, vy);
    float crossY = getCrossProduct(uz, vx, ux, vz);
    float crossZ = getCrossProduct(ux, vy, uy, vx);
```

```
 // normalize using cross product and length
 // divide by the sqrt of the product of the length of the vectors
 float total = (crossX * crossX) + (crossY * crossY) + (crossZ * crossZ);
 float length = sqrt(total);

 float normalX = crossX / length;
 float normalY = crossY / length;
 float normalZ = crossZ / length;

 //do what I commented out before, add each normalized value

 g_meshNormals[3 * indice1] += normalX;
 g_meshNormals[3 * indice1 + 1] += normalY;
 g_meshNormals[3 * indice1 + 2] += normalZ;

 g_meshNormals[3 * indice2] += normalX;
 g_meshNormals[3 * indice2 + 1] += normalY;
 g_meshNormals[3 * indice2 + 2] += normalZ;

 g_meshNormals[3 * indice3] += normalX;
 g_meshNormals[3 * indice3 + 1] += normalY;
 g_meshNormals[3 * indice3 + 2] += normalZ;
```

```
 for (int w = 0; w < g_meshNormals.size() / 3; ++w) {
     g_meshNormals[3 * w] = g_meshNormals[3 * w] / 6;
     g_meshNormals[3 * w + 1] = g_meshNormals[3 * w + 1] / 6;
     g_meshNormals[3 * w + 2] = g_meshNormals[3 * w + 2] / 6;
 }
```

External resources used:

http://www.opengl-tutorial.org/beginners-tutorials/tutorial-8-basic-shading/

Part 2: Make the teapot rotate

The rotation function that I implemented remembers the location during a pause and continues the rotation from that point. When the teapot is rotating, the time is stored in a start variable. When it stops, an end variable stores the stop time. When the teapot begins to rotate again, the end – start value is subtracted from the new getTime() value to give the original paused location. A huge help for this problem was the resource: https://open.gl/transformations which is an openGL resource and includes a section on transformations. There is a section about rotation around the Y-axis that helped to show what indexes matter and need to be changed:

Rotation around Y-axis:

$$\begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\theta \cdot x + \sin\theta \cdot z \\ y \\ -\sin\theta \cdot x + \cos\theta \cdot z \\ 1 \end{pmatrix}$$

Interestingly, the -sin value determined the direction of rotation. If I switched the sin(theta) and -sin(theta) values around, the teapot would change rotation from counter-clockwise or clockwise. I kept it as a counter-clockwise rotation.

```
void updateModelViewMatrix()
{
    clearModelViewMatrix();

    // TASK 2
    // The following code sets a static modelView matrix
    // This should be replaced with code implementing Task 2
    // You can use getTime() to change rotation over time
    if (teapotSpin) {
        //sin + or - determines clockwise or counter clockwise rotations
        //need to save the angle of where it is when paused, so that it doesnt skip around
        time = getTime() - pause;
        start = time;
        g_modelViewMatrix[0] = cos(time);
        g_modelViewMatrix[2] = -sin(time);
        g_modelViewMatrix[5] = 1.0f;
        g_modelViewMatrix[8] = sin(time);
        g_modelViewMatrix[10] = cos(time);
        g_modelViewMatrix[14] = -distance;
        g_modelViewMatrix[15] = 1.0f;
    }

    else
    {
        //need to preserve the time that the rotation was stopped at
        end = getTime();
        pause = end - time;

        g_modelViewMatrix[0] = cos(start);
        g_modelViewMatrix[2] = -sin(start);
        g_modelViewMatrix[5] = 1.0f;
        g_modelViewMatrix[8] = sin(start);
        g_modelViewMatrix[10] = cos(start);
        g_modelViewMatrix[14] = -distance;
        g_modelViewMatrix[15] = 1.0f;
    }
```

Part 3: Dolly Zoom Effect and Orthogonal Projection

Dolly Zoom:

The dolly zoom was probably the weirdest part of the assignment. I looked around trying to understand it better and found the resources: https://jsantell.com/3d-projection and https://www.mathematik.uni-marburg.de/~thormae/lectures/graphics1/graphics_6_1_eng_web.html#1. Both of these resources use the same idea of determining the distance by figuring out: halfWidth / tan(fov / 2); The hardest part about this idea was determining what fov was. I tried to simply determine fov by taking 60.f / 180 * M_PI, cos(60.f / 180 * M_PI), abs(cos(60.f / 180 * M_PI)), abs(cos(timer / 180 * M_PI)), and so many more things. In the end, what seemed to work was taking the cos, or sin, of the timer. If I didn't take the absolute value of this number, the dolly zoom would flip upside down every other cycle (I assumed this was something to do with negative numbers which is why I used the absolute value).

Also, no matter what I did to this, I could not get the teapot to not clip off the top of the screen and disappear for a frame when it reaches the top. I tried almost everything and couldn't get this to work correctly.

```
// Perspective Projection
if (enablePersp)
{
    // Dolly zoom computation
    if (enableDolly) {
        // TASK 3
        // Your code for dolly zoom computation goes here
        // You can use getTime() to change fov over time
        // distance should be recalculated here using the Equation 2 in the description file

        //not sure why but cos and sin here were pretty interchangeable
        fov = abs(sin(timer));
        distance = halfWidth / tan(fov / 2);

    }

    float fovInDegree = radianToDegree(fov);
    gluPerspective(fovInDegree, (GLfloat)g_windowWidth / (GLfloat)g_windowHeight, 1.0f, 40.f);
}
```

Orthogonal Projection:

The orthogonal projection wasn't too difficult once I understood it fully. I used the following resources:

https://lmb.informatik.unifreiburg.de/people/reisert/opengl/doc/glOrtho.html#targetText=DESCRIPTION,%2C%200%2C%200).%20- and https://www.learnopengles.com/tag/orthographic-projection/ .

I tried a lot of different projections, including setting all values to 1.f, setting the top and bottom based on the height, the widths, and several combinations of the width and the height. In the end, setting it to the height/ width and setting the top to be the negative value of this and the bottom to be the positive value worked the best. It was interesting to see how changing the number really messed up everything.

I wasn't quite sure if the teapot was supposed to get "bigger" or "smaller" when the button for orthogonal perspective was pressed. I found that the code below created a "bigger" orthogonal perspective and setting glOrtho(-1.5f, 1.5f, -1.f, 1.f, 1.f, 40.f); would create a "smaller" orthogonal perspective.

```
// Othogonal Projection
else
{
    // Scale down the object for a better view in orthographic projection
    glScalef(0.5, 0.5, 0.5);

    // TASK 3
    // Your code for orthogonal projection goes here
    // (Hint: you can use glOrtho() function in OpenGL)
    //https://lmb.informatik.uni-freiburg.de/people/reisert/opengl/doc/glOrtho.html#targetText=DESCRIPTION,%2C%200%2C%200).%20%2D
    //glOrtho(-1.f, 1.f, -1.f * (g_windowHeight), 1.f * (g_windowWidth), 1.f, 40.f);
    //glOrtho(-1.f, 1.f, -1.f , 1.f , 1.f, 40.f);
    glOrtho(-1.f, 1.f, -1.f * (float)(g_windowHeight) / (g_windowWidth), 1.f * (float)(g_windowHeight) / (g_windowWidth), 1.f, 40.f);

}
```

Part 4: Extra Credit

I really did my best to fully tackle this extra credit. I implemented the following extra features, all of which are included in the video:

Rotating Teapot Pause Remember:

As previously stated, the pause remember function works by "when the teapot is rotating, the time is stored in a start variable. When it stops, an end variable stores the stop time. When the teapot begins to rotate again, the end – start value is subtracted from the new getTime() value to give the original paused location."

```
if (teapotSpin) {
    //sin + or - determines clockwise or counter clockwise rotations
    //need to save the angle of where it is when paused, so that it doesnt skip around
    time = getTime() - pause;
    start = time;
    g_modelViewMatrix[0] = cos(time);
    g_modelViewMatrix[2] = -sin(time);
    g_modelViewMatrix[5] = 1.0f;
    g_modelViewMatrix[8] = sin(time);
    g_modelViewMatrix[10] = cos(time);
    g_modelViewMatrix[14] = -distance;
    g_modelViewMatrix[15] = 1.0f;
}

else
{
    //need to preserve the time that the rotation was stopped at
    end = getTime();
    pause = end - time;

    g_modelViewMatrix[0] = cos(start);
    g_modelViewMatrix[2] = -sin(start);
    g_modelViewMatrix[5] = 1.0f;
    g_modelViewMatrix[8] = sin(start);
    g_modelViewMatrix[10] = cos(start);
    g_modelViewMatrix[14] = -distance;
    g_modelViewMatrix[15] = 1.0f;
}
```

GL_SMOOTH and GL_FLAT:

Pressing the “Y” key will switch to flat, and pressing the “U” key will switch to smooth

```
if (p_key == GLFW_KEY_Y && p_action == GLFW_PRESS) {
    glClearColor(1.f, 1.f, 1.f, 1.0f);

    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glEnable(GL_DEPTH_TEST);
    glShadeModel(GL_FLAT);

}
```

```
if (p_key == GLFW_KEY_U && p_action == GLFW_PRESS) {
    glClearColor(1.f, 1.f, 1.f, 1.0f);

    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glEnable(GL_DEPTH_TEST);
    glShadeModel(GL_SMOOTH);
}
```

Change Colors:

The colors can be changed by pressing the ‘z’ key. The color can be changed and viewed with either the GL_SMOOTH or GL_FLAT shade models. The color changing is disable during the checkerboard display since the teapot was turning solid blue if the color was changed while the checkerboard was displayed. Otherwise, the color can be changed freely.

```
if (p_key == GLFW_KEY_Y && p_action == GLFW_PRESS) {
    glClearColor(1.f, 1.f, 1.f, 1.0f);

    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glEnable(GL_DEPTH_TEST);
    glShadeModel(GL_FLAT);

}
```