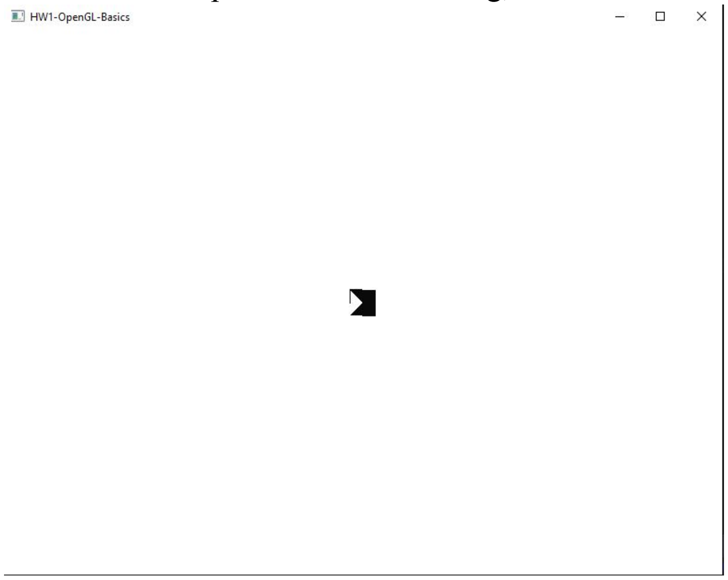
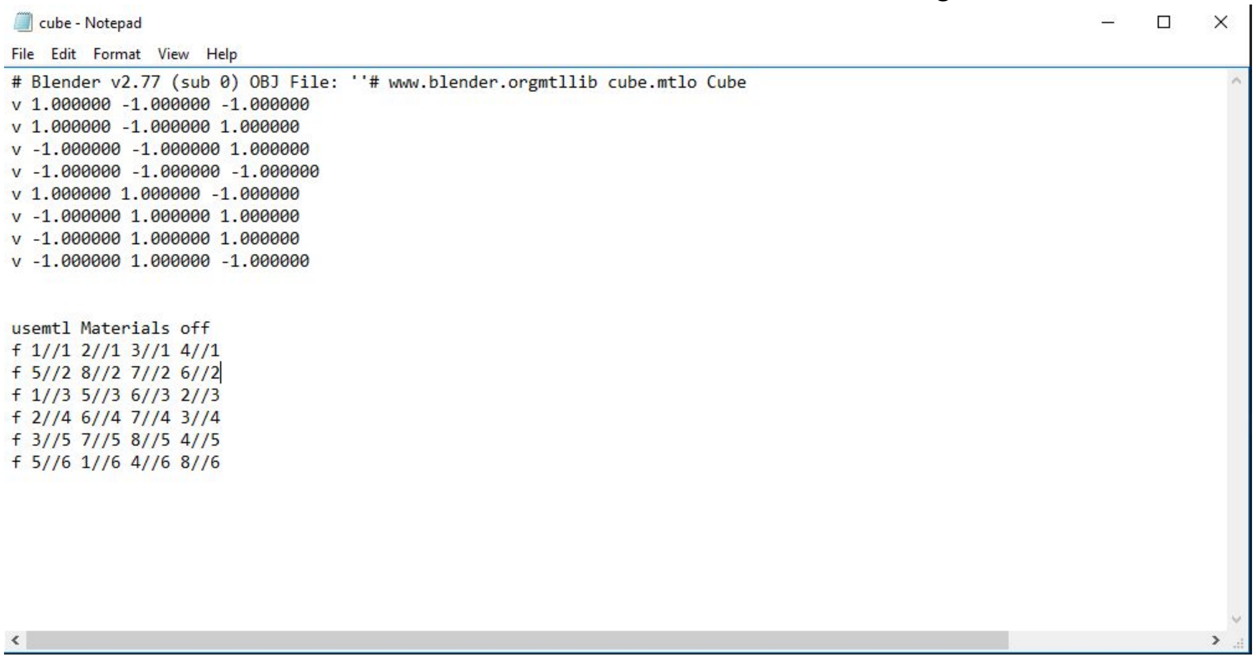
Cameron Jackson
CS 4600

HW1

The teapot model is pretty complex to work with, so one of the first things I did was export a basic cube from blender because I figured it would be easier to work with. However, when I tried to open it with the shading, this was the result:



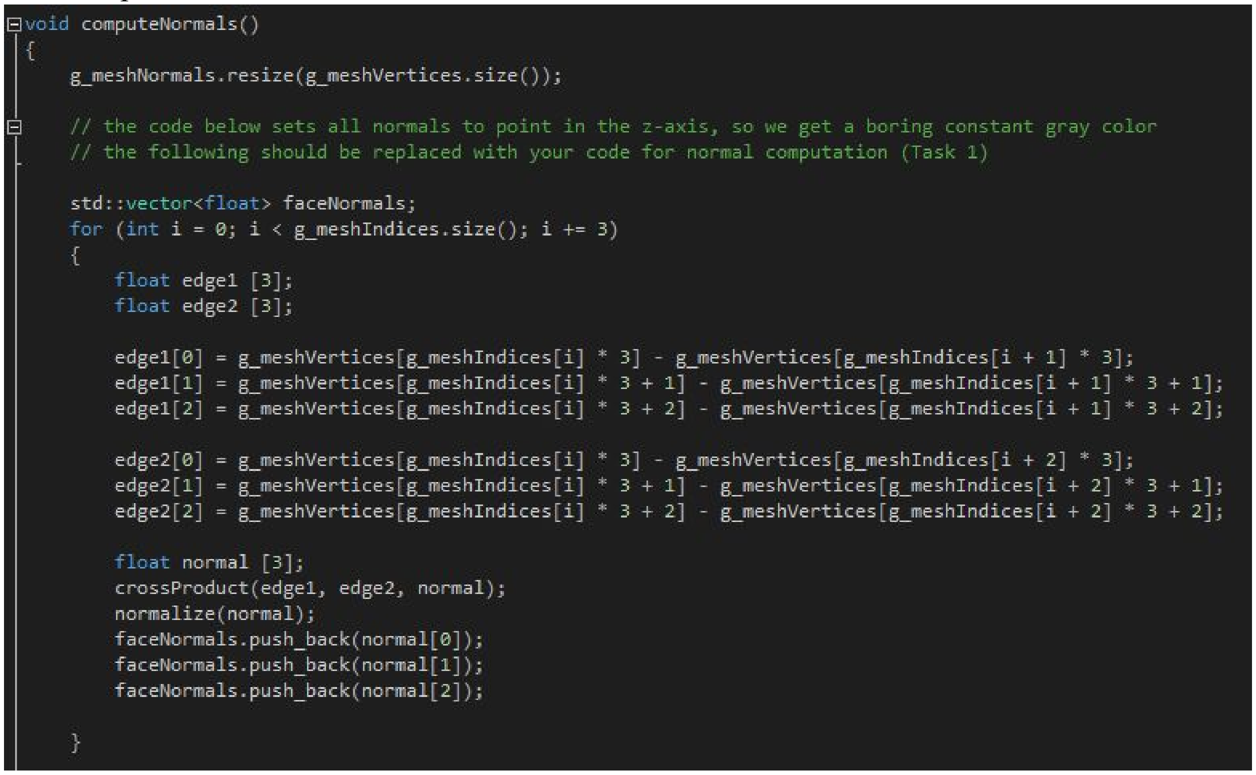
It turns out the file was stored in a different format than the one we were given:



```
# Blender v2.77 (sub 0) OBJ File: ''# www.blender.orgmtllib cube.mtlo Cube
v 1.000000 -1.000000 -1.000000
v 1.000000 -1.000000 1.000000
v -1.000000 -1.000000 1.000000
v -1.000000 -1.000000 -1.000000
v 1.000000 1.000000 -1.000000
v -1.000000 1.000000 1.000000
v -1.000000 1.000000 1.000000
v -1.000000 1.000000 -1.000000


usemtl Materials off
f 1//1 2//1 3//1 4//1
f 5//2 8//2 7//2 6//2
f 1//3 5//3 6//3 2//3
f 2//4 6//4 7//4 3//4
f 3//5 7//5 8//5 4//5
f 5//6 1//6 4//6 8//6
```

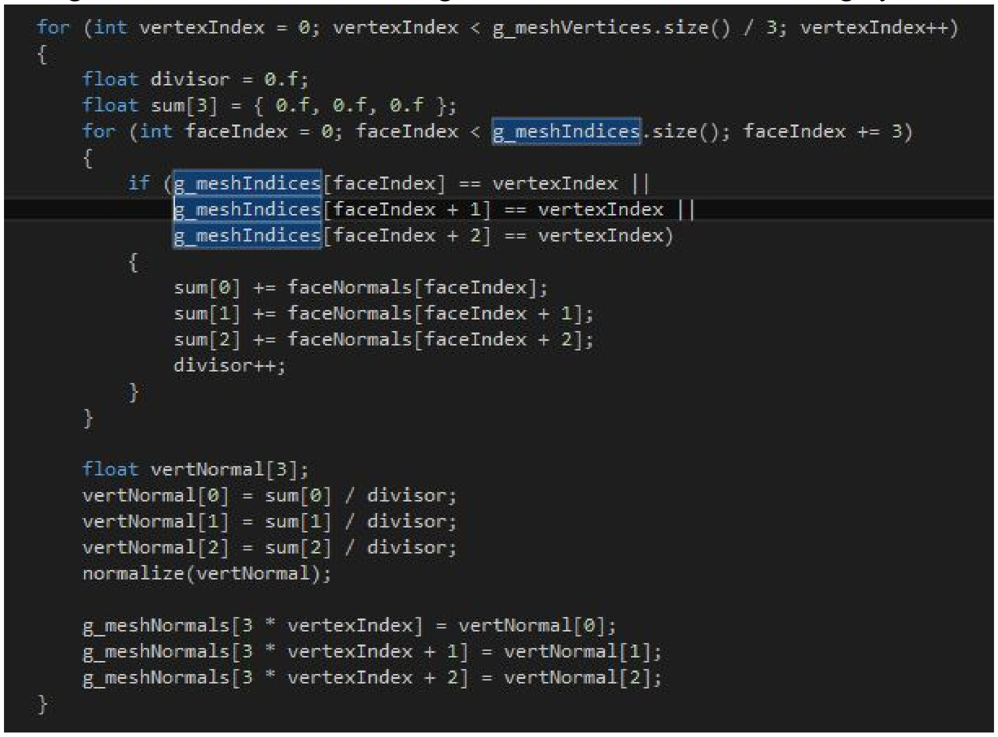
It looks like the indices of each vertex are individually parsed with an index of a face, instead of being grouped into threes to form a face. Kind of weird but not a huge deal.

In any case, I first calculated the face normals by iterating through each triplet of vertex indices and generating vectors for two of the three possible edges between them. The face normal is just the cross product of these vectors.



```
void computeNormals()
{
    g_meshNormals.resize(g_meshVertices.size());

    // the code below sets all normals to point in the z-axis, so we get a boring constant gray color
    // the following should be replaced with your code for normal computation (Task 1)

    std::vector<float> faceNormals;
    for (int i = 0; i < g_meshIndices.size(); i += 3)
    {
        float edge1 [3];
        float edge2 [3];

        edge1[0] = g_meshVertices[g_meshIndices[i] * 3] - g_meshVertices[g_meshIndices[i + 1] * 3];
        edge1[1] = g_meshVertices[g_meshIndices[i] * 3 + 1] - g_meshVertices[g_meshIndices[i + 1] * 3 + 1];
        edge1[2] = g_meshVertices[g_meshIndices[i] * 3 + 2] - g_meshVertices[g_meshIndices[i + 1] * 3 + 2];

        edge2[0] = g_meshVertices[g_meshIndices[i] * 3] - g_meshVertices[g_meshIndices[i + 2] * 3];
        edge2[1] = g_meshVertices[g_meshIndices[i] * 3 + 1] - g_meshVertices[g_meshIndices[i + 2] * 3 + 1];
        edge2[2] = g_meshVertices[g_meshIndices[i] * 3 + 2] - g_meshVertices[g_meshIndices[i + 2] * 3 + 2];

        float normal [3];
        crossProduct(edge1, edge2, normal);
        normalize(normal);
        faceNormals.push_back(normal[0]);
        faceNormals.push_back(normal[1]);
        faceNormals.push_back(normal[2]);

    }
```

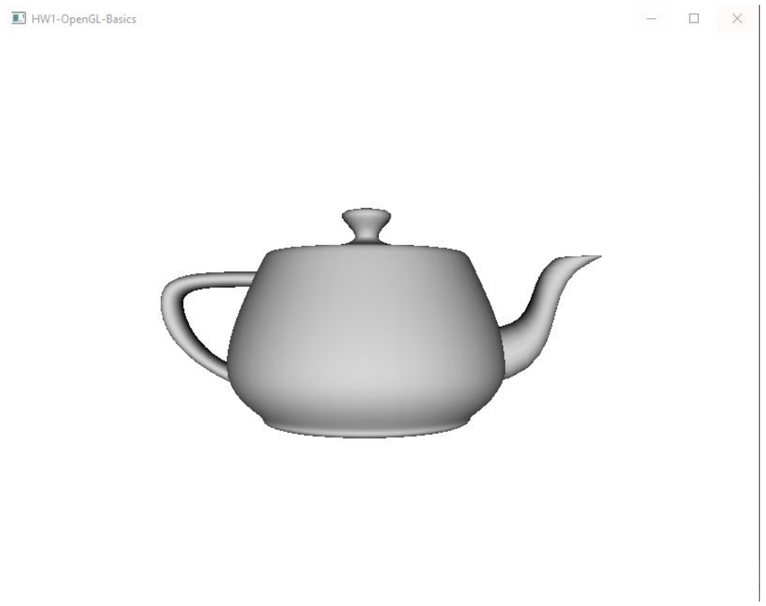
I then went through every vertex and averaged all the normals of its connecting faces by iterating through the face normals, summing the relevant ones, and dividing by the summed faces.



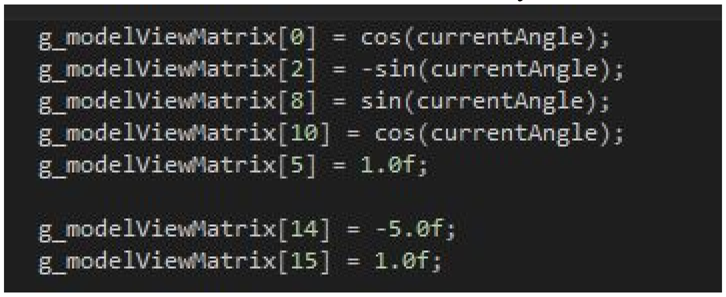
```
for (int vertexIndex = 0; vertexIndex < g_meshVertices.size() / 3; vertexIndex++)
{
    float divisor = 0.f;
    float sum[3] = { 0.f, 0.f, 0.f };
    for (int faceIndex = 0; faceIndex < g_meshIndices.size(); faceIndex += 3)
    {
        if (g_meshIndices[faceIndex] == vertexIndex ||
            g_meshIndices[faceIndex + 1] == vertexIndex ||
            g_meshIndices[faceIndex + 2] == vertexIndex)
        {
            sum[0] += faceNormals[faceIndex];
            sum[1] += faceNormals[faceIndex + 1];
            sum[2] += faceNormals[faceIndex + 2];
            divisor++;
        }
    }

    float vertNormal[3];
    vertNormal[0] = sum[0] / divisor;
    vertNormal[1] = sum[1] / divisor;
    vertNormal[2] = sum[2] / divisor;
    normalize(vertNormal);

    g_meshNormals[3 * vertexIndex] = vertNormal[0];
    g_meshNormals[3 * vertexIndex + 1] = vertNormal[1];
    g_meshNormals[3 * vertexIndex + 2] = vertNormal[2];
}
```

Admittedly, this is not the most efficient way to do this. It would probably be faster to store the indices of all the faces connected to a given vertex while the file is parsing so we don’t have to iterate again, but for our purposes, this is functional.

The Result:

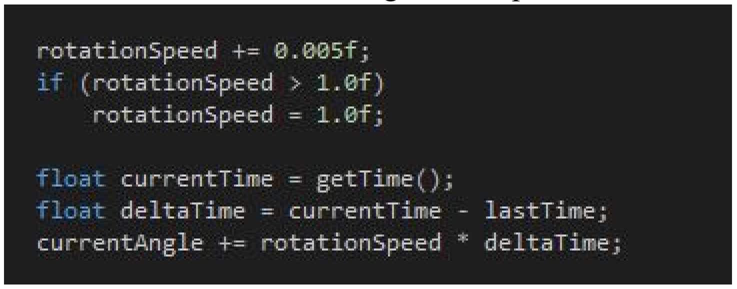


To make it rotate, I read up about transformation matrices from these tutorials:
http://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices/
http://www.opengl-tutorial.org/assets/faq_quaternions/index.html#Q26
And set a matrix for rotation around the y axis:



```
g_modelViewMatrix[0] = cos(currentAngle);
g_modelViewMatrix[2] = -sin(currentAngle);
g_modelViewMatrix[8] = sin(currentAngle);
g_modelViewMatrix[10] = cos(currentAngle);
g_modelViewMatrix[5] = 1.0f;

g_modelViewMatrix[14] = -5.0f;
g_modelViewMatrix[15] = 1.0f;
```

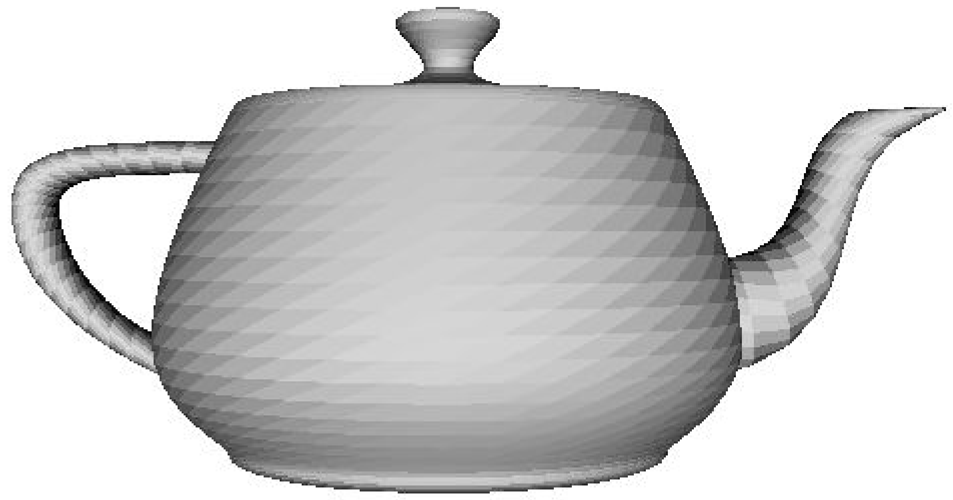
To make the rotation rate slowly increase, I started the rotation rate at 0 and increased a little each frame until it reached a max of 1. This is then multiplied by the change in time since the last frame and added to the angle of the previous frame to get a new angle.



```
rotationSpeed += 0.005f;
if (rotationSpeed > 1.0f)
    rotationSpeed = 1.0f;

float currentTime = getTime();
float deltaTime = currentTime - lastTime;
currentAngle += rotationSpeed * deltaTime;
```

Video:
https://www.youtube.com/watch?v=xUCguIeYTbE&feature=youtu.be

I also tried using the flat shading mode:



As far as I can tell, this just generates a single color value for every face using face normals instead of vertex normals. This makes every pixel of a face have the same color, so it appears flat. The smooth shading model, however, seems to calculate every pixel individually by interpolating between the vertex normals of a face. In other words, the closer a pixel is to a vertex, the more similar it is to the color of its normal. This allows the model to appear much more realistic than the raw model may seem.