CAP4730 Model Transformations Report

Jonathan Bryan

March 2024

1 Introduction

In this project I use OpenGL to load Wavefront .obj files and render them to the screen. The object is then manipulated using translation, rotation, and scaling matrices before being drawn to the screen. User input comes in the form of rotation using the mouse cursor or keys, rotation using the arrows keys, and scaling using the mouse scroll wheel.

2 Geometry Data Representation

Our models are stored in object files which have a list of vertices and a list of faces. The vertices are referenced in a list of faces, which create polygons. These polygons can be parsed into triangles which are drawn to the screen. The algorithm for converting a polygon with n vertices to triangles is as follows:

1. Make a vector of all vertices.
2. Create a triangle of the first, second to last, and last vertices in the vector.
3. Pop the last vertex from the vector.
4. Repeat until only 2 vertices left.

2.1 Separate Triangles

The simplest method of creating triangle geometry is by storing each of the points on the triangle separately. Storing three vertices per triangle can be redundant, however, because neighboring triangles share vertices.

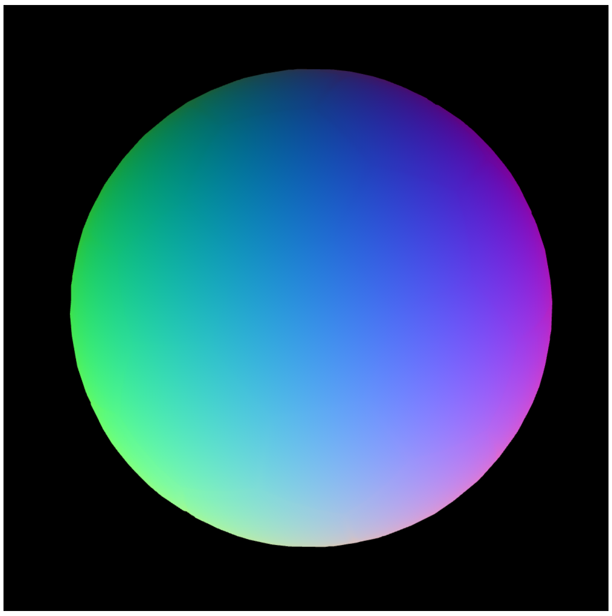
2.2 Indexed Triangles

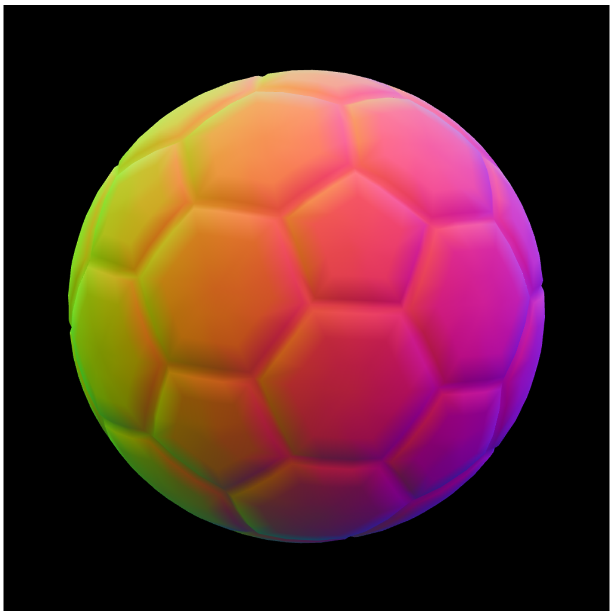
We can leverage the fact that our object files list each vertex once, and faces simply store an index to vertices. We can directly implement indexed triangles by keeping a master list of vertices and creating a list of triangle indices using the 4-step method described above.

2.3 Storing Position and Color

Our triangle vertex data is stored in a vertex buffer object, but we can store other things there as well. The first way of storing other data is placing all of the data for a triangle and parsing the information as it is passed to the vertex shader. This method allows us to create different vertex attributes and give them different sizes and locations. The second way of storing the data is creating a new vertex buffer object. This method simplifies editing the data within the buffer because different components such as position and color are stored separately. For this project I opted to use the second method and create multiple vertex buffer objects.

For the coloring of the objects I used two methods. In the first method the color is based on the normalized location of the vertex. The second uses the loaded vertex normals from the object file. We can see in the example below that the normal data gives us texturing and the illusion of depth. In a more advanced program we would utilize these normals to shade the objects.


Color based on position.

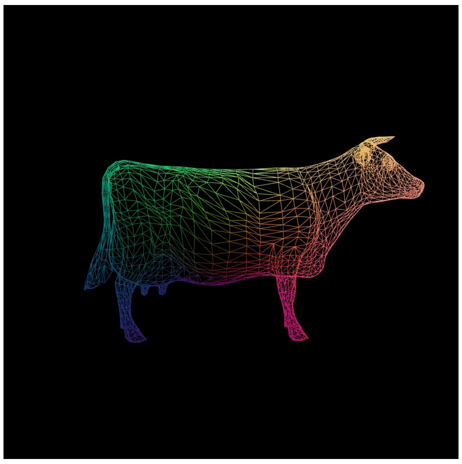

Color based on normal.

3 OpenGL Shaders

Shaders allow us to define how each part of the rasterization process happens using the GPU. This specialized GLSL code is usually stored as a string and compiled into a shader program which OpenGL uses. There are many different kinds of shaders but for this project we only need the vertex and fragment shaders.

3.1 Vertex Shader

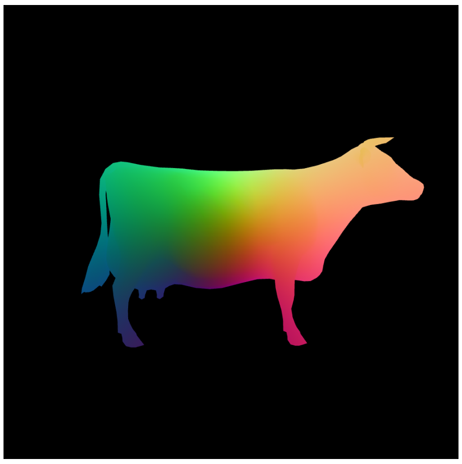
In this project the vertex shader file is named "source.vs" and gets parsed into the application via the Shader class. The vertex shader allows us to define where vertices are placed on the screen before we dive into the per pixel fragment shader. The input of the shader is the two data attributes color and position. We also pass in a "uniform" which is a global value shared by all vertices. The output of this shader is gl_Position describes the screen coordinate of the pixel and the color of the vertex which comes directly from the color VBO and gets passed to the fragment shader.



Wireframe cow model showing vertices.

3.2 Fragment Shader

In this project the fragment shader file is named "source.fs" and gets parsed into the application via the Shader class. The fragment shader interpolates between vertices and has more granular control over the final color of each pixel. The input of the shader is the color passed in by the vertices. The output is a vec4 which describes the RGBA value of the pixel.



Full cow model showing wireframe filled in.

4 Model View Matrix

Our model view matrix is responsible for the translation, rotation, and scaling of the entire mesh. This matrix is multiplied by each vertex on either the GPU or CPU.

4.1 Translation

The translation component of the model view matrix uses homogeneous coordinates to shift the mesh in the x, y or z direction. For this project we limit the movement to the x and y directions using the arrow keys. Below is an example translation which would shift the mesh right 2 and down 0.5:

$$\begin{bmatrix} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & -0.5 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

4.2 Rotation

The rotation component of the model view matrix allows us to spin the object around different axis. In this project the W and S keys or vertical mouse movement rotate the object around the x-axis, the A and D keys or horizontal mouse movement rotate the object around the y-axis, and the Q and E keys rotate around the z-axis. To make the control feel more normal the axis are taken from the object orientation instead of global world coordinates. Below is a 45° rotation around the z-axis:

$$\begin{bmatrix} 0.707 & 0.707 & 0 & 0 \\ -0.707 & 0.707 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

4.3 Scaling

The scale component of the model view matrix moves all points further or closer to the center of the model. In this project the scroll wheel controls the scale matrix. Below is a matrix that will scale the model by 3:

$$\begin{bmatrix} 3 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

4.4 Applying Transformation on GPU vs CPU

For this section I will use data collected when loading the cow.obj mesh. I chose this mesh because it has a large number of triangles so the different methods will have a significant impact on performance. The most obvious method of transforming our mesh is multiplying each vertex by our model view matrix. We can do this multiplication in a loop and refresh our vertex buffer object each frame. When using this method The more efficient way of applying these transformations to our model is multiplying the components on the CPU and sending them to our GPU for per vertex multiplication. This process is done by sending a mat4 uniform to the vertex shader.

FPS	EBO	GPU or CPU
200	No EBO	CPU
775	EBO	CPU
1700	No EBO	GPU
1750	EBO	GPU

We can see that using the GPU makes the largest improvement in performance. The EBO had a large improvement on the CPU because we had to iterate over a much smaller list of vertices and there is less duplicate data.

5 References

5.1 ImGui

ImGui (github.com/ocornut/imgui) was utilized in this project for the menu. The interactive manual (pthom.github.io/imgui_manual_online) was helpful when implementing the matrix table representation.

5.2 LearnOpenGL

I read through parts of the LearnOpenGL (learnopengl.com/) tutorials for this project. Code written to store and load shader files came from the "Shaders" tutorial (learnopengl.com/Getting-started/Shaders). The "Hello Triangle" section (learnopengl.com/Getting-started/Hello-Triangle) helped me in implementing the EBO.