Project 10 Vertex and Fragment Shaders

Audrey Nelson

Project Description:
Alter the rendered appearance of a given sphere using the Vertex and Fragment shaders from your computers Graphics Processing Unit.

Why use Vertex and Fragment Shaders?
It is generally a good idea to use vertex and fragment shaders to display textures rather than modifying the texture data and mesh vertices in a programming language such as processing because it is much faster. For example, if you were trying to render something like an ocean and wanted to see the waves it would be very expensive to be constantly changing a mesh that represented that ocean each frame so that the viewer could see waves. Instead, using the vertex shader, you can keep the original mesh for the ocean and simply change the way that mesh is displayed. Because the vertex shader is part of the GPU it would be much faster and less expensive because it is done directly on the hardware. Similarly, when trying to convey texture, it would be very expensive to make a brick wall model with every small crack you may want to see; so instead, you can place a texture on it and then alter the perceived locations of each vertex using the vertex shader as well as use the fragment shader to add shadows in the correct places to produce the look of depth on an actually flat wall.

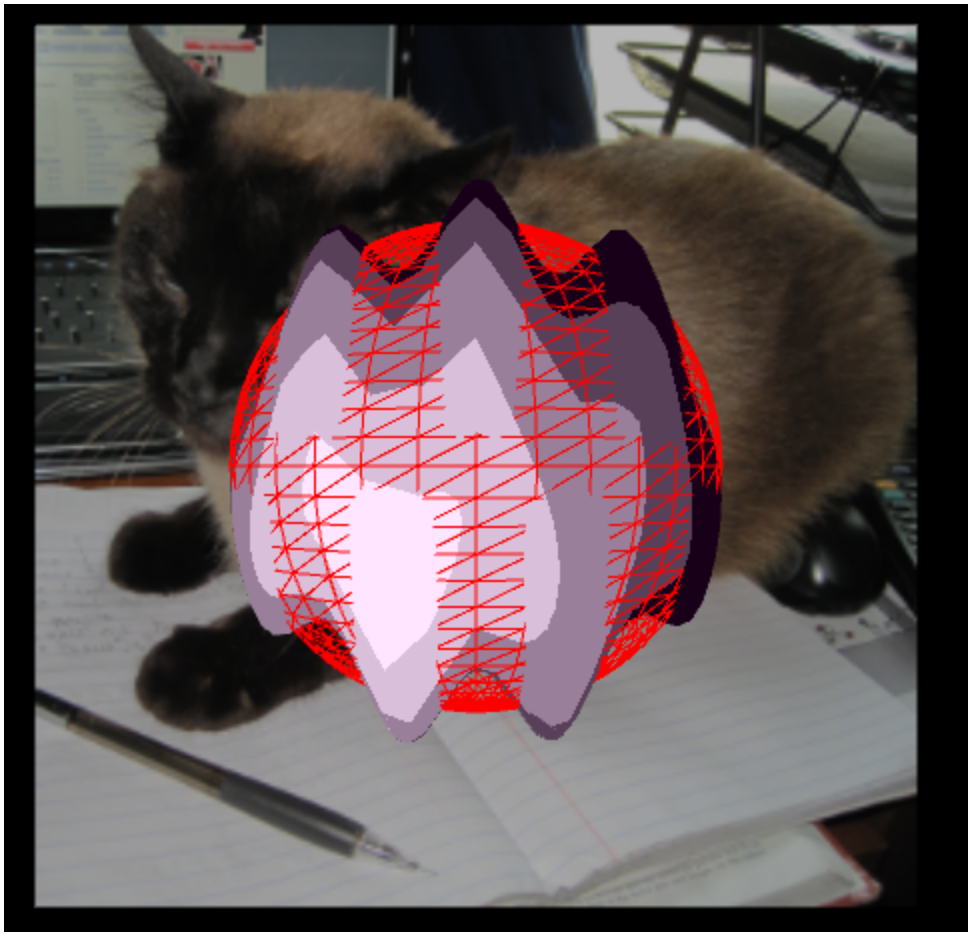
Pairing 1:
Fragment Shader:
The first fragment shader I created was cel-shading, also known as toon shading. For this shader I created thresholds for the intensity of the diffuse lighting giving the shading a more blocky appearance. (*note: I also tinted the color to be slightly purple)

```
if(color.r <0.3) color.rgb = vec3(0.1, 0.0, 0.1);
  else if(color.r <0.55) color.rgb = vec3(0.35, 0.25, 0.35);
  else if(color.r < 0.8) color.rgb = vec3(0.6, 0.5, 0.6);
  else if(color.r < 0.95) color.rgb = vec3(0.85, 0.75, 0.85);
  else color.rgb = vec3(1.0, 0.9, 1.0);
```

Vertex Shader:
For my first vertex shader I created a warbling effect of the y position following a sin wave and using the mouse position to animate it. To create the sin wave, I added the scaled value of the sin of the x position (converted to radians) of the vertex to the y position of the vertex. To

animate the sin wave, I added the x value of the mouse location to the x value of the current vertex before taking the sin.

```
vert.y += 20.0*sin(5.0 * 3.1471 * (vert.x + mouseX) / 180.0);
```



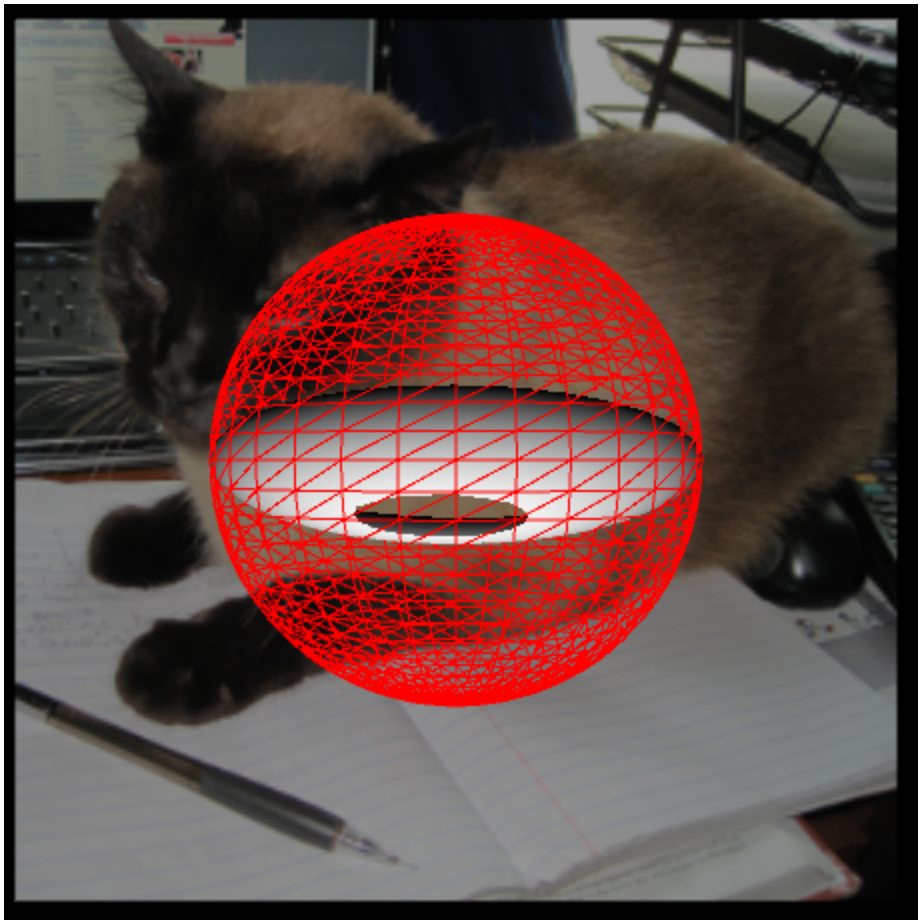
Pairing 2:
Fragment Shader:
For my second fragment shader I created an x-ray light that made the sphere transparent where the light was brightest. For this shader I also made the sphere transparent where no light reached it so that you could see all the way through the sphere to the background. To do this, I changed the alpha values of all pixels will light intensity over 0.95 or with no light intensity to be 0.0.

```
if(color.r > 0.95 || color.r == 0.0) color.a = 0.0;
```

Vertex Shader:

For my second vertex shader, I squashed the ball as a function of the mouse position. To do this, I moved each vertex's y component closer to the center of the ball by my "squash factor" which was determined by the y position of the mouse on the screen.

```
vert.y += ((0.0 - vertex.y)/squash);
```



Pairing 3: Additional Points

Fragment Shader:

For my third fragment shader, I made the ball pulse by changing its color as a smooth function of time. To do this, I had a time variable that counted up from 0 to 1 and then back down to 0. I then set the red value of each pixel to be equal to the time while the green value was always 1. This produced a gradual fading between green and yellow.

```
color.r = time;
color.g = 1.0;
```

Vertex Shader:

For my third vertex shader, I made the ball smaller by moving all the points closer to the center and then translated the ball around the screen based on the x,y position of the mouse. To do this, I simply calculated the distance from the mouse to the center of the screen and scaled this distance, I then added the scaled translation factor to all vertices of the sphere.

```
vert.y += ((0.0 - vertex.y)/2.0) + yOff;
vert.x += ((0.0 - vertex.x)/2.0)+ xOff;
```