Intro to WegGL: Vertex & Fragment Shaders

February 28, 2019

This document provides an introduction to WebGL using Vertex & Fragment shaders. First fragment shaders are introduced using an online editor. An example of rendering a circle using only fragment shaders is examined. Second, vertex shaders with fragment shaders are introduced using a locally hosted webpage. Two examples of vertex/fragment shader combinations are examined. At the end of the document, the reader should understand how vertex and fragment shaders are connected and used.

1 Requirements

WebGL provides an OpenGL implementation for webpages that will use hardware acceleration for rendering. In order to verify that your browser, os, and hardware stack are working correctly: please load the following web page `http://glslsandbox.com` and select a shader example. You should see an image that is rendered using WebGL and the select fragment shader. This webpage will be used for the first

A locally hosted webpage will be used for the second examples. All html, js, obj, and png files are provided here: `https://www.dropbox.com/s/braoyji4ne6im57/ProjectG.zip?dl=0`. In order to run the webpage, open a console in the directory with the provided file. Run the following command to start a local host webserver:

- Python2: python -m SimpleHTTPServer 12345
- Python3: python -m http.server 12345

Open a web browser and navigate to the following webpage: `http://localhost:12345/projectg.html` and validate that the example is working.

2 Shaders

The OpenGL pipeline takes a vertex specification, such as model geometry, and a series of shaders (small programs written in shader language), starting with the vertex shader and ending with the fragment shader, to output pixels. At minimum a fragment shader must be specified to render pixels, and model geometry requires both a vertex shader and fragment shader. Additional shader layers that will not be covered, provide transformations to tessellation (subdivision of vertices), geometry manipulation, and the catchall compute shader.

Working backwards through the pipeline each pixel is rendered by a fragment shader, this will be examined first to create a simple circle on the screen and calculate shaded lighting on sphere. Interesting overlay effects can be achieved in the fragment shader, simple shaded coloring, textures, rippling luminosities etc. The first example fragment shaders applied to the whole screen. The second example applies a different fragment shader to the waves and teapot. Creating complex lighting, color, and effects in a screen requires the composition of multiple fragment shaders applied to the results of multiple vertex shaders.

Two vertex/fragment shaders are then examined. The first vertex shader simply passed geometry directly though and is paired with a scaled lighting fragment shader. The second vertex shader creates waves on a surface and is paired with a texture mapping fragment shader.

3 Fragment Shader: Circle & Sphere Lighting

The example Circle and Sphere Lighting shaders can be found at `http://glslsandbox.com/e#52974.0` and `http://glslsandbox.com/e#52975.0` respectively.

Normally fragment shaders are used to calculate color, texture, and lighting information to produce a final pixel color. For the example Circle shader, the distance from the center of the screen is calculated and pixels are colored either red or grey. Assigning gl_FragColor in the fragment shader is mandatory to render any pixel. There are other built in variables in the fragment shader spec that are not examined.

The surfacePosition variable is bound by the GLSLsandbox program, the screen can then zoom and scroll over the surface. The starting screen to surface position is from (-1,-1) to (1,1) with (0,0) at the center. The circle is drawn by calculated the distance from the center, and either rendering the pixel red if the distance is less then the radius of the circle, or grey otherwise.

The Sphere lighting shader starts with the circle shader and calculates a lighting model for each pixel. This example assumes there is a sphere located at (0,0,0) with radius 0.5, and a fixed light position at (2,2,2). At each pixel/surfacePosition, the sphere surface point (X,Y,Z) is solved. Then the light direction vector to the sphere surface point is used to solve the shading to apply to the color red.

These two examples demonstrate the basics of fragment shading to deposit colors in gl_FragColor. The next two examples have simple fragment shaders to calculate light shading and use texture maps. Complex lighting, coloring, and texturing are achieved in fragment shaders, its even possible to create the illusion of surface geometry in this layer, as demonstrated by the Sphere lighting example.

4 Vertex & Fragment Shader: Textured Waves

The next vertex/fragment shaders are located in projectg.html. The remaining code is to load geometry and textures, bind them, compile the shaders, and link them together in the program to create the rendered image.

The fragment shader is a simple texture mapping shader. The variable vTextureCoord is output by the vertex shader and uSampler bound in the calling program. The function simply maps the location of uSampler to the texture location, and colors the fragment with the texture color by setting gl_FragColor.

The vertex shader for the waves takes a flat plane for geometry and transforms the vertex based on the distance from the origin. aPosition is the original vertex location in the plane geometry and transforms the location with sin/cos. The transformed vertex is moved and positioned by multiplying by the uPMatrix and uMVMatrix matrix respectively, these are bound variables in the calling program. The final vertex position is assigned by setting gl_Position, this is mandatory for a vertex shader.

5 Vertex & Fragment Shader: Teapot

The vertex shader for the teapot simply transforms the teapot vertex geometry to the position specified by uMVMatrix, uNMatrix, and uPMatrix. These matrix transforms are variables that are bound in the calling program. The heavy work of constructing the matrices to transform by is done in the calling program. This vertex shader has simply moved the teapot geometry to the correct location.

The fragment shader for the teapot colors the surface of the teapot with different intensity colors based on a light source. Starting with the vertex position from the vertex shader, a 4d vertex: normalize and dividing by w to calculate the surface position in world coordinates. The light position and surface position direction are used to calculate a shading/intensity value. Depending on the intensity calculated, a color value is selected to color the gl_Fragment.

Implementing a smoothly colored & shaded surface can be quickly achieved by changing how the gl_FragColor is calculated. Currently the intensity value is used to select from 4 color values. Selecting a single color instead such as vec4 red_color = vec4(1.0,0.0,0.0,0.0). Then scale the colors values by the intensity, such as red_color.x = red_color.x * intensity, lastly set gl_FragColor =red_color to set the smoothly shaded color.

6 Shading Pipeline

The reader should now have an understanding of how the new opengl pipeline works. The calling program loads vertex & texture data, binds them, compiles shaders, then composes them together to create a rendered scene. Only vertex & fragment shaders have been examined, the first and last stage of the opengl pipeline respectively.