Lab 3 - Fragment Shaders

Intro
In Lab 2 you used a vertex shader to transform 3D shapes. But the shapes were just rendered as solid colors! Wouldn’t it be lovely if we could do something more interesting?

Fragment Shaders
Enter a second kind of shader: the **fragment shader**. In the same way that a vertex shader runs on every vertex to output a position, a fragment shader runs on every *pixel* (of every polygon) to output a *color*. (For the purposes of this lab we will use the terms “fragment” and “pixel” interchangeably.) Here’s our good ol’ OpenGL pipeline again so you can see where fragment shaders fit in.

A vertex shader and a fragment shader are combined to create something called a **shader program**, which is required to draw anything with OpenGL. There are other types of shaders (tessellation and geometry) that can be added to a shader program as well, but they’re optional (which is why they’re not in the diagram above).

---

1 What’s the difference between a “fragment” and a “pixel”? For one, the output of a fragment shader (i.e. a fragment) is not necessarily written to a pixel in the rendered image. This could happen if you’re rendering a polygon in the background polygon that’s behind another one that has already been drawn. Also it is possible to use multisampling, which runs multiple fragment shaders per pixel and averages the results, in which case there is not a one-to-one relationship between fragments and pixels.
The function for creating a shader program from a vertex and fragment shader is already written for you in ResourceLoader::createShaderProgram(...), which returns the new shader program’s ID.

Often you’ll have multiple shader programs to render different objects differently. You can change which shader program you’re using by passing a program ID to glUseProgram.

Before we start writing fragment shaders, let’s set up a simple square to test them on.

**Task 1:**
In GLWidget::initializeGL(), initialize m_square to be a square in the XY-plane, where x and y go from -0.5 to 0.5. Feel free to use any drawing mode you like (GL_TRIANGLES, GL_TRIANGLE_STRIP, or GL_TRIANGLE_FAN). In paintGL(), set the active program to m_solidProgramID and draw the square. After drawing, unbind the shader program by calling glUseProgram(0). You should see a white square on the screen.

The shader program referenced by m_solidProgramID uses the shaders solid.vert and solid.frag. Take a look at solid.vert. All it does is set gl_Position to the position attribute. We’re not using 3D in this lab so we don’t have to worry about the model, view and projection matrices.

Now take a look at solid.frag; it should look very similar to the vertex shader. Instead of outputting the color to a built-in variable like gl_Position, we output to a custom “out” variable, which we have named fragColor. Currently the fragment shader just outputs white: (r, g, b) = (1, 1, 1).

For this shader program, we would like to send a color as a uniform to the shader, and set each pixel to that color. Uniforms work exactly the same in fragment and vertex shaders, so this will work just like the uniforms in Lab 2.

**Task 2:**
In solid.frag declare a “color” uniform and set the fragColor to the uniform’s value. In GLWidget::paintGL(), set the uniform’s value to a color other than white. Remember that uniforms need to be set after the shader program is bound and before the shape is drawn. You should now see the square rendered as a different color.

**Multiple Vertex Attributes**
Now let’s make our square a little more interesting by making a color gradient.
To do this, we would like to assign each vertex a color, as shown above, and have the pixels *interpolate* their color between these vertices.

How do we assign a color to each vertex? We'll treat color as another *vertex attribute* just like position. This means we’re going store the vertex colors in a VBO, and tell the VAO how the color attribute is stored. One way we could do this would be to create separate VBOs for vertex positions and vertex colors, like so:

\[
\begin{array}{cccc}
X_1 & Y_1 & Z_1 & X_2 \\
R_1 & G_1 & B_1 & R_2 \\
\end{array}
\quad \begin{array}{ccc}
Y_2 & Z_2 & \cdots \\
G_2 & B_2 & \cdots \\
\end{array}
\]

Instead, we’re going to *interleave* the positions and colors in the same VBO. This ends up being the fastest option, because all the attributes for a given vertex are stored contiguously in memory.
We need to inform the VAO about how the attributes are interleaved. We can do this using the last two arguments of glVertexAttribPointer (which is called by OpenGLShape::setAttribute). For a given attribute:

- **pointer** indicates the offset in bytes from the beginning of the VBO’s data store to the first vertex
- **stride** indicates the length in bytes from the beginning of one vertex to the beginning of the next vertex.

Up until now we always set pointer and stride to 0, which OpenGL takes to mean “tightly packed.” When stride is greater than 0, OpenGL uses the definitions above.

Let’s add the color data to our VBO, and then we’ll discuss how to use it in the fragment shader.

**Task 3:**
Interleave positions and colors in the array used to initialize `m_square`. Now you will need two calls to `setAttribute`. Use 1 as the index for the color attribute. Run your program and make sure the square still shows up.

**Vertex Shader Output**
Let’s turn our attention to `gradient.vert` and `gradient.frag`. Since we just added a new vertex attribute, let’s declare a new input in the vertex shader.

**Task 4:**
In `gradient.vert`, add a new “in” variable for the color attribute. Set its location to 1.

Another use for vertex shaders besides transforming vertex positions is to send outputs to the fragment shader. An output variable is declared as an “out” in the vertex shader and an “in” in the fragment shader. For each pixel in a triangle, the fragment shader interpolates the output values from the triangle’s vertices.
**Task 5:**
- In gradient.vert, declare a new “out vec3” to send the color attribute to the fragment shader. Set its value to the vertex attribute’s value. (Note that the vertex shader’s out variable has to have a different name from its in variable.)
- In gradient.frag, make a new “in vec3” with the same name as the vertex shader’s output. Set the pixel color to that value.
- In GLWidget::paintGL(), set the active shader program to m_gradientProgramID and draw the square in the appropriate switch case.

You should now see a gradient when you run your program! (The color interpolation may look slightly different depending on how you tessellated the square.)

**Textures**

Now let's move on to texture.vert and texture.frag. We’ll use these shaders to render a texture, or image, on the square.

The first step is to load a texture into OpenGL. A texture object is created and bound using glGenTextures and glBindTexture, with GL_TEXTURE_2D as the target. You can send image data to the bound texture using glTexImage2D. Here are some notes about how to use its arguments in this lab:

- Set “level” to 0
- Set “internalformat” and “format” to GL_RGBA
- Set “type” to GL_UNSIGNED_BYTE
- Set “data” to image.bits()
- And yes, you read the documentation correctly. There is a parameter that “must be 0”…

The last thing we need to do is set two texture parameters. The image we are using is 585 pixels by 585 pixels. Depending on the size of the textured object that gets rendered, the original image will likely have to be scaled up or down. OpenGL has a few built-in ways to scale images, and you’re required to specify which one you want before you can use the texture.

The parameters that control the scaling technique are called GL_TEXTURE_MAG_FILTER and GL_TEXTURE_MIN_FILTER, for scaling up (MAGnifying) and scaling down (MINifying). These are set using glTexParameteri. For now, just set them to GL_NEAREST.

**Task 6:**

In GLWidget::initializeGL(), initialize a texture object using m_textureID and the provided QImage. Again, the relevant functions are glGenTextures, glBindTexture, glTexImage2D, and glTexParameteri. (Make sure the texture is bound to GL_TEXTURE_2D before calling glTexImage2D or glTexParameteri.)
Before moving to the shaders, we need to add another vertex attribute to the VBO. Each vertex corresponds to a different corner of the image. To specify the corner for each vertex, we will use **UV-coordinates**. In UV-coordinates (0, 0) refers to the top left of the image and (1, 1) is the bottom right.

![UV-coordinates diagram](image)

**Task 7:**
Interleave UV-coordinates along with positions and colors in your VBO. Adjust your calls to `setAttribute` accordingly. Use 2 as the attribute index and note that the “size” of the new attribute is 2, not 3. Run your program and make sure the first two shader programs still work.

Your VBO should now be structured something like this:

```
X_1  Y_1  Z_1  R_1  G_1  B_1  U_1  V_1  X_2  Y_2  Z_2  R_2  G_2  B_2  U_2  V_2  ...
```

Now it’s time to move to texture.vert and texture.frag. In this shader program we want to interpolate the UV-coordinates across all pixels. That way each pixel has a unique set of UV-coordinates and can look up the correct **texel** in the texture. (A texel is a pixel in the texture image. The distinction is made so as not to confuse this with pixels on the screen.)

We need a way to access the texture from the fragment shader. In GLSL the type `sampler2D` represents a texture. Since we’re using the same image for the entire object, you will declare it as a uniform.

Unlike the other uniforms we’ve seen, you do not have to call `glUniform` to set the texture uniform’s value. Instead, make sure the texture is bound to `GL_TEXTURE_2D` when you draw.
OpenGL will automatically send this texture to the sampler2D uniform declared in the fragment shader.

**Task 8:**
Use `texture.vert` and `texture.frag` to interpolate UV-coordinates across the image. In the fragment shader, declare a sampler2D uniform. Set the output color to the corresponding texel by calling the GLSL function “texture” which uses the sampler2D and the uv-coordinates. In `GLWidget::paintGL()`, bind the texture to `GL_TEXTURE_2D` and draw using this shader program in the last switch case. You should see an image displayed on the square!

Also, remember that this is C++, where you have to clean up your resources. In this case, we have to delete the texture we gave to OpenGL. Luckily, there’s a handy `glDeleteTexture` function just for this! Call it in your `GLWidget`’s destructor.

Once you’ve got all three shader programs working, have a TA check off your lab!