Reflection
CPSC 501 - Advanced Programming Techniques

Dylan Leclair

October 5, 2020

0.1 Shaders

Shaders are written in GLSL, a C-like language.

GLSL is tailored for use with graphics and contains useful features specifically targeted at vector and matrix manipulation.

Shaders always begin with a version declaration, followed by a list of input and output variables, uniforms (common between all instances) and output the results in its output variables.

A shader has the typical structure:

```
#version version_number
in type in_variable_name;
in type in_variable_name;

out type out_variable_name;

uniform type uniform_name;

void main()
{
  // process input(s) and do some weird graphics stuff
  ...
  // output processed stuff to output variable
  out_variable_name = weird_stuff_we_processed;
}
```

In the vertex shader, each input variables is also known as a *vertex attribute*. There is a maximum number of vertex attributes we're allowed to declare limited by the hardware. OpenGL guarantees there are always at least 16 4 components vertex attributes available, but some hardware may allow for more.

0.1.1 Types

GLSL has all the basic types, but also two container types: `vectors` and `matrices`.

A vector in GLSL is a 1,2,3 or 4 component container for any of the basic types just mentioned.

They can take the following form (n represents the number of components):

- vecn: the default vector of n floats
- bvecn: a vector of n booleans
- ivecn: a vector of n integers
- uvecn: a vector of unsigned integers
- dvecn: a vector of double components

For general purposes, the basic version is sufficient!

Use the variables x,y,z,w to access the four components. GLSL allows you to use rgba for colors or stpq for texture coordinates - accessing the same components.

0.1.2 Ins and outs

Shaders are great, but they are best when they work together. Each shader can specify inputs and outputs using those keywords and wherever an output variables matches with an input variable of the next shader stage they're passed along.

The vertex shader should receive some sort of input otherwise it would be pretty ineffective. The vertex shader differs in its input, in that it receives its input straight from the vertex data. To define how the vertex data is organized, we specify the input variables with location metadata so we can configure the vertex attributes on the CPU.

We did this with `layout (location = 0)` in the last part. The vertex shader requires an extra layout specification for it's inputs so we can link it with the vertex data.

The other exception is that the fragment shader requires a vec4 color output variable, since the fragment shaders need to generate a final output color. If you fail to specify an output color in your fragment shader, the color buffer output for those fragments will be undefined!

If we want to send data from one shader to the other, we'd have to declare an output in the sending shader and a similar input in the receiving shader.

When the types and the names are equal on both sides, OpenGL will link those variables together and its possible to send data between shaders!

0.1.3 Uniforms

Uniforms are another way to pass data from our application of the CPU and the shaders on the GPU. Uniforms are however slightly different compared to vertex attributes. First of all, uniforms are **global**. A uniform variable is *unique per shader program object*, and **can be accessed from any shader at any stage in the shader program**.

Second, whatever you set the uniform value to, uniforms will **keep their values until they're reset or updated**.

To declare a uniform in GLSL, we simply add the uniform keyword to a shader with a type and a name. From that point on we can use the newly declared uniform in the shader.

If you declare a variable that is not used in your shader, GLSL will remove it! Keep this in mind when looking for errors.

To access a uniform value from CPU code, there is a bit of a process:

1. Get the index/location of the uniform attribute in our shaders
2. Update its values

```
float timeValue = glfwGetTime();
float greenValue = (sin(timeValue) / 2.0f) + 0.5f;
int vertexColorLocation = glGetUniformLocation(shaderProgram, "ourColor"); // ourColor is the name o
glUseProgram(shaderProgram);
glUniform4f(vertexColorLocation, 0.0f, greenValue, 0.0f, 1.0f);
```

The above is a simple example of a shader that will change the color of a shape over time!

If glGetUniformLocation returns a -1, it could not find the location!

Note that finding the uniform location does not require toy to use the shader program first, but updating a uniform **does** require you to first use the program since it will perform actions on the **currently active** shader program.

Since OpenGL is a C library, note that you will have to use different variants of the glUniform function to change different types of data.

```
// position attribute
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);
// color attribute
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)(3* sizeof(float)));
glEnableVertexAttribArray(1);
```

The vertex buffers have to be slightly reconfigured to support the addition of colours. All you need to do is make sure that your VAO is set up so that the VBOs are interpreted correctly!

We only supplied 3 colors, but the triangle is beautifully shaded! This is because of *fragment interpolation*.

0.2 A note on preprocessor directives

```
#ifndef SHADER_H
#define SHADER_H

#endif
```

Using these informs the compiler to only include and compile this header file if it hasn't been included yet, even if multiple files include the shader header. This prevents linking conflicts.