3D Graphics Programming
Lab 3: Experimenting with Shaders

Karsten Pedersen
Department of Creative Technology

September 13, 2018

Now that we have a triangle rendering we can start experimenting with the shader code. We currently are displaying a blue triangle. We know that if we change it in the fragment shader to be a different colour, we can make it any colour we need. However, using this method it would be hard to draw a number of differently coloured triangles. We would need a different shader for each one! This is where *Uniforms* come in.

Uniforms

You can think of a *Uniform* as a variable within the shader that you can change from your *C/C++* code. In order use a *Uniform* we need to modify the respective shader object. Since we want to set the colour via our *Uniform*, we will be editing the *Fragment Shader*.

```
const GLchar *fragmentShaderSrc =
  "uniform vec4 in_Color;" \
  "void main()" \
  "{" \
  "  gl_FragColor = in_Color;" \
  "}" \
  "";
```

We add a new ***vec4*** variable (RGBA) to hold the colour and we label it as ***uniform*** (in a similar way to how we did the triangle position attribute). We then use the colour directly rather than the hard coded blue value. If you try to run the program now you will probably see a black triangle. This is because the uniform is at the default value of ***(0, 0, 0, 0)*** (Black). So what we now need to do is set the uniform value.

The first step is to find the uniform from the shader (**After** the shader has been linked via ***glLinkProgram***, unlike when you were binding the attribute location).

```
// Store location of the color uniform and check if successfully found
GLint colorUniformId = glGetUniformLocation(programId, "in_Color");

if(colorUniformId == -1)
{
  throw std::exception();
}
```

Note:
You will notice that we are using a ***GLint*** rather than a ***GLuint***. This means it is a signed integer and the main reason for this is that unlike with the shader, the value of ***0*** is valid (i.e the first uniform) and instead ***-1*** is given on an error. Only signed integral types are able to hold a negative value correctly.

With all this in place, at any point within the program we can change the colour used by the shader. The following code demonstrates this and you should end up with output similar to **Figure 1**.

```
// Bind the shader to change the uniform, set the uniform and reset state
glUseProgram(programId);
glUniform4f(colorUniformId, 0, 1, 0, 1);
glUseProgram(0);
```

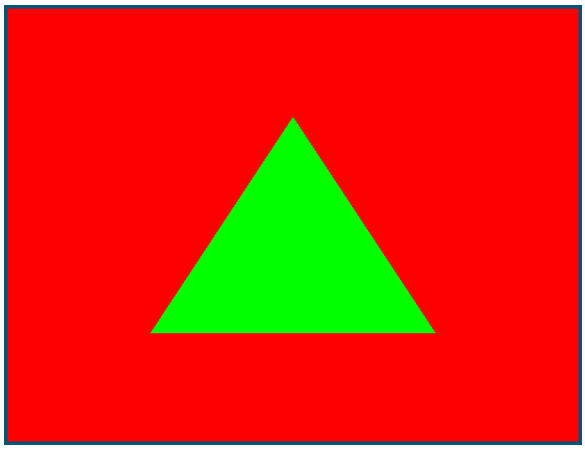

Figure 1: *A screen shot of our triangle with its colour set via a Uniform*

Note:
Unlike obtaining the uniform location via ***glGetUniformLocation***, the ***glUniform4f*** function does not take an argument of the shader program to operate on. This is a bit messy because instead you need to bind and unbind the current shader to operate on via ***glUseProgram***. This state machine driven approach is one of the main complaints developers have with *OpenGL* over something like i.e *DirectX*. In practice it is just something you have to get used to and make sure via disciplined coding you keep resetting the state back to a known state. Once you have written helper functions or even a game engine on top of *OpenGL*, it will become much easier.

Attributes

If we wanted each point of the triangle to have a different colour, light value or texture coordinate (such as in **Figure 2**), a *Uniform* would not be viable because that remains constant during the drawing of the entire shape rather than per vertex. Instead we use the exact same technique as we did for the positions of the triangle in the first place using *Attribute streams*.

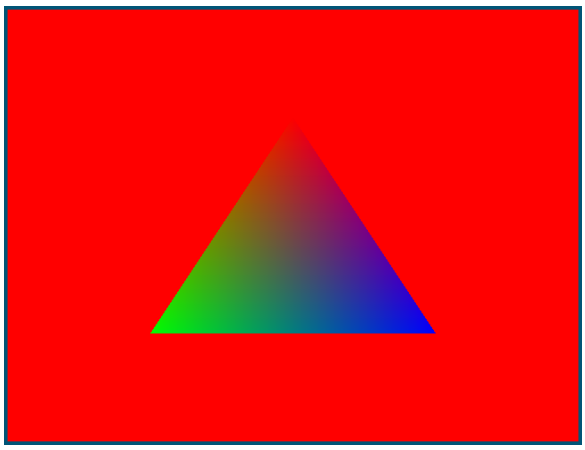

Figure 2: *A screen shot of our triangle with its colour set via an Attribute Stream*

One slight complexity with *Attributes* is that they always need to be passed in to the *Vertex shader* and passed through into the *Fragment shader* so that they can be interpolated between the two. For this reason we will need to replace the *Vertex shader* with the following:

```
const GLchar *vertexShaderSrc =
  "attribute vec3 in_Position;" \
  "attribute vec4 in_Color;" \
  "" \
  "varying vec4 ex_Color;" \
  "" \
  "void main()" \
  "{" \
  "  gl_Position = vec4(in_Position, 1.0);" \
  "  ex_Color = in_Color;" \
  "}" \
  "";
```

Note:
Remember that now we have modified the shader code and removed the uniform variable (because we are going to use an attribute stream instead), your bit of code that looks for the uniform variable via *glGetUniformLocation* will now return as an error. Since we no longer need it, you must delete that too.

Here you will see that we have an additional *Attribute* called ***in_Color*** that we pass to another variable of type ***varying*** via the simple assignment in the ***main*** function. This is all that is required to pass a variable through from the *Vertex shader* to the *Fragment shader*. Next we will simply use this value from within the *Fragment shader* as the final output colour. The following listing shows this simple process.

```
const GLchar *fragmentShaderSrc =
  "varying vec4 ex_Color;" \
  "void main()" \
  "{" \
  "  gl_FragColor = ex_Color;" \
  "}" \
  "";
```

Note:
The variable name ***ex_Color*** **must** be the same in each shader, otherwise *OpenGL* does not know how the two variables link together.

Next, as we did with out positions *Attribute stream* we define the colours of each point of the triangle (red, green, blue).

```
const GLfloat colors[] = {
  1.0f, 0.0f, 0.0f, 1.0f,
  0.0f, 1.0f, 0.0f, 1.0f,
  0.0f, 0.0f, 1.0f, 1.0f
};
```

We then do the process of creating the new *VBO*, uploading the data onto it and assigning it to position ***1*** of the *VAO*.

```
GLuint colorsVboId = 0;

// Create a colors VBO on the GPU and bind it
glGenBuffers(1, &colorsVboId);

if(!colorsVboId)
{
  throw std::exception();
}

glBindBuffer(GL_ARRAY_BUFFER, colorsVboId);

// Upload a copy of the data from memory into the new VBO
glBufferData(GL_ARRAY_BUFFER, sizeof(colors), colors, GL_STATIC_DRAW);

// Bind the color VBO, assign it to position 1 on the bound VAO
// and flag it to be used
glBindBuffer(GL_ARRAY_BUFFER, colorsVboId);

glVertexAttribPointer(1, 4, GL_FLOAT, GL_FALSE,
  4 * sizeof(GLfloat), (void *)0);

glEnableVertexAttribArray(1);
```

Note:
Take extra notice of the seemingly magic numbers. We use ***1*** rather than ***0*** because our colour buffer is the second position of the *VAO*. We also use ***4*** instead of ***3*** because our RGBA colour buffer is a ***vec4*** rather than a ***vec3***.

Before linking your *Shader program*, remember to instruct *OpenGL* that you want to connect the *in_Color Attribute* variable with the second *Attribute stream* (the colours) within the *VAO*.

```
glBindAttribLocation(programId, 1, "in_Color");
```

You should now have a colourful triangle appearing when you run the program. If you find that you are getting different results then very carefully go through the code and check it. There is a lot of similar code here and it is very easy to make a mistake. We will look at a slightly more scalable architecture in the next lab.

Note:
So far we have covered using ***glUniform4f*** to assign a ***vec4*** to the shader program. Try using others such as ***glUniform1f*** in order to assign a ***uniform float***. See if you can assign a *uniform* in the *Vertex shader* in order to get the triangle to move left or right.