COM3503/4503/6503: 3D Computer Graphics

Dr Steve Maddock

Exercise sheet 2: Shaders

This exercise sheet will follow Joey de Vries's tutorial section on Shaders: https://learnopengl.com/#!Getting-started/Shaders.

1 Introduction

We'll look at the basics of shaders, including how to transfer data to the vertex and fragment shaders. Later exercise sheets will extend this when we cover Lighting. However, we will not do advanced shader work in this module (as is covered in Joey's sections on Advanced OpenGL and Advanced Shading). Joey covers GLSL in his tutorial. Instead, I'll focus on the changes we need to make to accommodate JOGL.

2 Ins and outs (attributes)

Program: Shader01.java. This program is similar to the programs we looked at in Exercise sheet 1. As in previous programs, the main programs sends position attributes to the vertex shader, as indicated by the keyword in. The main difference is that the colour for the triangle is set in the vertex shader rather than in the fragment shader. The colour attribute is then passed from the vertex shader, through the GPU's rasterization process, to the fragment shader. This is indicated by the use of the out and in keywords in Program listing

```
// ***************************************************
/* THE SHADER
 */

private String vertexShaderSource =
  "#version 330 core\n" +
  "\n" +
  "layout (location = 0) in vec3 position;\n" +
  "\n" +
  "out vec4 vertexColor;\n" +      // vertexColor passed to rasterisation process
  "\n" +
  "void main() {\n" +
  "  gl_Position = vec4(position.x, position.y, position.z, 1.0);\n" +
  "  vertexColor = vec4(0.55f, 0.0f, 0.55f, 1.0f);\n" +
                                  // vertexColor set here - magenta
  "}";

private String fragmentShaderSource =
  "#version 330 core\n" +
  "\n" +
  "out vec4 color;\n" +
  "\n" +
  "in vec4 vertexColor;\n" +       // vertexColor received from rasterisation process
  "\n" +
  "void main() {\n" +
  "  color = vertexColor;\n" +     // output colour for fragment set to input colour
  "}";
```

Program listing 2.1: Using the in and out keywords to pass data from the vertex shader to the fragment shader

2.1, which gives the vertex and fragment shader sources. We're still using hard-coded String variables for the shaders. Later on, we'll load these from file instead.

The rasterization process takes the values defined at each vertex of the triangle (position and vertexColor attributes) and fills the inside of the triangle with a series of fragments (which are then rendered to the pixels of the screen display). The values from the vertices are bilinearly interpolated over the surface of the triangle – we'll discuss this process in lectures. We set the colour attribute of a vertex in the vertex shader. Each vertex sent by the main program (received by the vertex shader using 'in position') is given the same colour value by the vertex shader. Program listing 2.2 gives the data that is sent to the GPU from the CPU, in this case three vertices which make one triangle. Since the colour attribute of each vertex is the same, the whole triangle will be coloured in the same value, i.e. every fragment is given the same value. In a later program we'll see how to give each vertex a different colour.

It is worth noting that the attributes being set in the vertex shader are per-vertex attributes. The position attribute is different for each instance of the shader and the vertexColor attribute, in this example, is the same for every instance of the vertex shader. Remember, on the GPU there are multiple instances of the shader running in parallel, one per thread (where each core on a GPU can execute multiple threads), each dealing with a different vertex. That's what makes the GPU so powerful.

```
private float[] vertices = {
   0.0f,  0.5f, 0.0f,  // Top Middle
   0.5f, -0.5f, 0.0f,  // Bottom Right
  -0.5f, -0.5f, 0.0f   // Bottom Left
};
private int[] indices = {
    0, 1, 2
};
```

Program listing 2.2: The data for one triangle

3 Uniforms

Program: Shader02.java. Uniforms are used to pass data from the CPU application to the shaders on the GPU. A uniform can be accessed by either the vertex or fragment shader in the shader program object. The same uniform constant is available to every instance of the vertex and fragment shaders launched by a render call (e.g. glDrawElements()). Uniforms can then be changed before subsequent render calls.

In Shader02_GLEventListener.java, a uniform is declared in the fragment shader (see Program listing 3.1). The main application now needs to be changed to send a value to the fragment shader. Program listing 3.2 shows the necessary changes.

The uniform declared in the fragment shader is accessed using `glGetUniformLocation()` with the shaderProgram and the name of the uniform supplied as parameters. `glUniform4f` can then be used to set the new values of the uniform, by supplying the 4 float values required, as indicated by the name of the method. The list of vertices is then drawn as usual.

In this example, rather than send the same colour value every time the triangle is rendered (60 times a second), the green value is changed every time the render method is called by using the system time (see Program listing 3.3).

At this point, it is worth noting that there are only two ways to pass information to shaders: uniforms and per-vertex attributes, which we'll look at in the next section.

```
private String vertexShaderSource =
  "#version 330 core\n" +
  "\n" +
  "layout (location = 0) in vec3 position;\n" +
  "\n" +
  "void main() {\n" +
  "  gl_Position = vec4(position.x, position.y, position.z, 1.0);\n" +
  "}";

private String fragmentShaderSource =
  "#version 330 core\n" +
  "\n" +
  "out vec4 color;\n" +
  "\n" +
  "uniform vec4 ourColor;\n" +     // ourColor received from main application
  "\n" +
  "void main() {\n" +
  "  color = ourColor;\n" +
  "}";
```

Program listing 3.1: A uniform is declared in the fragment shader

Exercise

1. Experiment with changing each component of the colour.
2. Add a uniform to the vertex shader that takes an x and a y value and then use these values to move the triangle. (Hint: You'll need a different version of glUniform*. Simple addition can be done in the vertex shader by adding the uniform (x,y) to the vertex position, e.g. `position.x+x`

```java
  public void render(GL3 gl) {
    gl.glClear(GL.GL_COLOR_BUFFER_BIT | GL.GL_DEPTH_BUFFER_BIT);

    double elapsedTime = getSeconds() - startTime;

    gl.glUseProgram(shaderProgram);

    float redValue = 0.9f;
    float greenValue = (float)Math.sin(elapsedTime*5);
    float blueValue = 0.2f;
    int vertexColourLocation = gl.glGetUniformLocation(shaderProgram, "ourColor");
    gl.glUniform4f(vertexColourLocation, redValue, greenValue, blueValue, 1.0f);

    gl.glBindVertexArray(vertexArrayId[0]);
    gl.glDrawElements(GL.GL_TRIANGLES, indices.length, GL.GL_UNSIGNED_INT, 0);
    gl.glBindVertexArray(0);
  }
```

Program listing 3.2: The render method is updated to set the uniform in the fragment shader

```java
public void init(GLAutoDrawable drawable) {
    //...rest of method...
    startTime = getSeconds();
  }

private double startTime;

  private double getSeconds() {
    return System.currentTimeMillis()/1000.0;
  }
```

Program listing 3.3: Methods to get the system time

4 More attributes

Program: Shader03.java. In the last section, we looked at using a uniform to change the vertex colour. However, this changed the colour of every vertex. Instead, we'd like to be able to give each vertex a different colour. We can do this by adding extra attributes to the vertex data. Joey's tutorial describes the process in detail. I'll concentrate on the JOGL-related changes.

Program listing 4.1 shows the changes to the vertex data structure. Three extra attributes are added to each vertex. These are the red, green and blue values of the colour at a vertex. I've also introduced three class attributes that describe the vertex data and which are used when dealing with the buffers. Before looking at these, we'll examine the shaders – see Program listing 4.2.

```
private float[] vertices = {
   0.0f,  0.5f, 0.0f, 1.0f, 0.0f, 0.0f, // Top Middle, red
   0.5f, -0.5f, 0.0f, 0.0f, 1.0f, 0.0f, // Bottom Right, green
  -0.5f, -0.5f, 0.0f, 0.0f, 0.0f, 1.0f  // Bottom Left, blue
};

private int vertexStride = 6;
private int vertexXYZFloats = 3;
private int vertexColourFloats = 3;

private int[] indices = {
  0, 1, 2
};
```

Program listing 4.1: Changes to the vertex data structure

```
private String vertexShaderSource =
  "#version 330 core\n" +
  "\n" +
  "layout (location = 0) in vec3 position;\n" +
  "layout (location = 1) in vec3 color;\n" +
  "out vec3 ourColor;\n" +
  "\n" +
  "void main() {\n" +
  "  gl_Position = vec4(position, 1.0);\n" +
  "  ourColor = color;\n" +
  "}";

private String fragmentShaderSource =
  "#version 330 core\n" +
  "in vec3 ourColor;\n" +
  "out vec4 color;\n" +
  "\n" +
  "void main() {\n" +
  "  color = vec4(ourColor, 1.0f);\n" +
  "}";
```

Program listing 4.2: The shaders

```
private void fillBuffers(GL3 gl) {
  gl.glGenVertexArrays(1, vertexArrayId, 0);
  gl.glBindVertexArray(vertexArrayId[0]);
  gl.glGenBuffers(1, vertexBufferId, 0);
  gl.glBindBuffer(GL.GL_ARRAY_BUFFER, vertexBufferId[0]);
  FloatBuffer fb = Buffers.newDirectFloatBuffer(vertices);

  gl.glBufferData(GL.GL_ARRAY_BUFFER, Float.BYTES * vertices.length,
                  fb, GL.GL_STATIC_DRAW);

  int stride = vertexStride;
  int numXYZFloats = vertexXYZFloats;
  int offset = 0;
  gl.glVertexAttribPointer(0, numXYZFloats, GL.GL_FLOAT, false,
                           stride*Float.BYTES, offset);
  gl.glEnableVertexAttribArray(0);

  int numColorFloats = vertexColourFloats;
  offset = numXYZFloats*Float.BYTES;
  gl.glVertexAttribPointer(1, numColorFloats, GL.GL_FLOAT, false,
                           stride*Float.BYTES, offset);
  gl.glEnableVertexAttribArray(1);

  gl.glGenBuffers(1, elementBufferId, 0);
  IntBuffer ib = Buffers.newDirectIntBuffer(indices);
  gl.glBindBuffer(GL.GL_ELEMENT_ARRAY_BUFFER, elementBufferId[0]);
  gl.glBufferData(GL.GL_ELEMENT_ARRAY_BUFFER, Integer.BYTES * indices.length,
                  ib, GL.GL_STATIC_DRAW);
  gl.glBindVertexArray(0);
}
```

Program listing 4.3: Changes to fillBuffers()

The vertex shader is changed to receive the colour value in attribute location 1. Thus it will expect the main application to send it a colour value for each vertex in the same way that we have been sending a position value for each vertex. As in previous examples, that colour per vertex is then sent through the rasterization process to the fragment shader. However, since the colour at each vertex is different, the fragments of the triangle will all be different colours (based on interpolating the values at the vertices).

The next step is to fill the buffers with the vertex attribute data. Program listing 4.3 shows the relevant method. The key part is highlighted in bold. The first part is the same as previous programs and describes how the vertex position data is transferred to attribute location 0 in the vertex shader. The variables declared in Program listing 4.1 are used. The stride variable states how big each set of attributes for a vertex is, in this case 6 floats. The numXYZFloats is 3 and the numColorFloats is 3. The xyz position data is stored at offset 0 in the vertices data, i.e. it is the first 3 floats in each set of 6 floats for a vertex. The offset is changed to 3*Float.BYTES when describing the colour data, since this data comes after the 3 floats. The colour data is transferred to attribute location 1 in the vertex shader.

Program listing 4.4 gives the render method. This is unchanged from previous programs. All the work has been done in the fillBuffers method to set up the description of how to transfer data to the GPU.

```
public void render(GL3 gl) {
  gl.glClear(GL.GL_COLOR_BUFFER_BIT | GL.GL_DEPTH_BUFFER_BIT);

  gl.glUseProgram(shaderProgram);

  gl.glBindVertexArray(vertexArrayId[0]);
  gl.glDrawElements(GL.GL_TRIANGLES, indices.length, GL.GL_UNSIGNED_INT, 0);
  gl.glBindVertexArray(0);
}
```

Program listing 4.4: render()

5 Dynamically changing the VBO data

Program: Shader04.java. This example shows how to change the data for a vertex whilst the program is running. A similar approach could be used to change a subset of a collection of vertices. Program listing 5.1 gives a method to change the xyz position value of a single vertex and a method to change the colour value of a single vertex. The first method uses the index of the vertex in the list multiplied by the stride to access the specific vertex in the list of vertex data. The second method uses a modified version that takes into account that the colour values come after the xyz position values in the set of values for a single vertex. Thus vertexXYZFloats (= 3, the number of floats in the xyz position data) is required. The render method is updated accordingly to make use of the method to change the values (see Program listing 5.2). Note: In fillbuffers(), glBufferData() has been using GL.GL_STATIC_DRAW when setting up the buffer for the vertex data. Now that the data is changing with every call to render, this is changed to GL.GL_DYNAMIC_DRAW[1].

Exercise

Experiment with changing the xyz position and rgb colour values of the vertices. I've included some commented out lines in Shader05_GLEventListener to help you get started.

```
private void replaceVBO_XYZ(GL3 gl, int index, float x, float y, float z) {
  float[] aVertex = {x,y,z};
  gl.glBindBuffer(GL.GL_ARRAY_BUFFER, vertexBufferId[0]);
  FloatBuffer fb = Buffers.newDirectFloatBuffer(aVertex);
  gl.glBufferSubData(GL.GL_ARRAY_BUFFER,
                     Float.BYTES * index * vertexStride,
                     Float.BYTES * aVertex.length, fb);
  gl.glBindBuffer(GL.GL_ARRAY_BUFFER, 0);
}

private void replaceVBO_RGB(GL3 gl, int index, float x, float y, float z) {
  float[] aVertex = {x,y,z};
  gl.glBindBuffer(GL.GL_ARRAY_BUFFER, vertexBufferId[0]);
  FloatBuffer fb = Buffers.newDirectFloatBuffer(aVertex);
  gl.glBufferSubData(GL.GL_ARRAY_BUFFER,
                     Float.BYTES * (index * vertexStride + vertexXYZFloats),
                     Float.BYTES * aVertex.length, fb);
  gl.glBindBuffer(GL.GL_ARRAY_BUFFER, 0);
}
```

Program listing 5.1: Changing the xyz position and colour values of a single vertex in the relevant GPU buffer

[1] There is some argument as to whether or not it is necessary to change GL.GL_STATIC_DRAW to GL.GL_DYNAMIC_DRAW as modern hardware/drivers can automatically manage this.

```
  public void render(GL3 gl) {
    gl.glClear(GL.GL_COLOR_BUFFER_BIT | GL.GL_DEPTH_BUFFER_BIT);
    double elapsedTime = getSeconds() - startTime;
    replaceVBO_XYZ(gl, 0, (float)Math.sin(elapsedTime), (float)Math.cos(elapsedTime), 0);
    replaceVBO_XYZ(gl, 1, (float)Math.sin(elapsedTime)*0.5f,
                          (float)Math.cos(elapsedTime)*0.5f, 0);
    replaceVBO_XYZ(gl, 2, (float)Math.cos(elapsedTime*0.5),
                          (float)Math.sin(elapsedTime*0.5), 0);
    gl.glUseProgram(shaderProgram);
    gl.glBindVertexArray(vertexArrayId[0]);
    gl.glDrawElements(GL.GL_TRIANGLES, indices.length, GL.GL_UNSIGNED_INT, 0);
    gl.glBindVertexArray(0);
  }
```

Program listing 5.2: render()

6 Shader class

Program: Shader05.java. In all the previous programs, the code to set up the shaders has remained unchanged, even if the source for the vertex and fragment shaders has changed. It makes sense to separate the shader code out into a separate class and load the shaders from file. The new class is Shader.java (see Program listing 6.1). The new code is then used in Shader05_GLEventListener.java and the text files vs5.tx and fs5.txt are used for the vertex and fragment shader source code, respectively. The majority of the code in Program listing 6.1 is the same as previous examples, namely method compileAndLink(). The constructor loads the source code from the relevant text files. The class could now be extended in the same way as Joey's class to include methods to set particular uniform values in the shaders. I'll leave this an exercise. Shader05_GLEventListener.java makes use of the Shader class. A private variable called shader of type Shader is declared as an attribute of the class and an instance of this is created in method initialise(). Method render() can then use the shader with the call shader.use(gl);

Exercises

1. Adjust the vertex shader so that the triangle is upside down.
2. Specify a horizontal offset via a uniform and move the triangle to the right side of the screen in the vertex shader using this offset value.

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.nio.charset.Charset;
import com.jogamp.opengl.*;
import com.jogamp.opengl.util.glsl.*;

public class Shader {

  private static final boolean DISPLAY_SHADERS = false;
  private int shaderProgram;
  private String vertexShaderSource;
  private String fragmentShaderSource;

  public Shader(GL3 gl, String vertexPath, String fragmentPath) {
    try {
      vertexShaderSource = new String(Files.readAllBytes(Paths.get(vertexPath)),
                                      Charset.defaultCharset());
      fragmentShaderSource = new String(Files.readAllBytes(Paths.get(fragmentPath)),
                                        Charset.defaultCharset());
    }
    catch (IOException e) {
      e.printStackTrace();
    }
    if (DISPLAY_SHADERS) display();
    shaderProgram = compileAndLink(gl);
  }

  public void use(GL3 gl) {
    gl.glUseProgram(shaderProgram);
  }
```

Program listing 6.1: Shader.java

```
  private void display() {
    if (DISPLAY_SHADERS) {
      System.out.println("***Vertex shader***");
      System.out.println(vertexShaderSource);
      System.out.println("\n***Fragment shader***");
      System.out.println(fragmentShaderSource);
    }
  }

  private int compileAndLink(GL3 gl) {
    String[][] sources = new String[1][1];
    sources[0] = new String[]{ vertexShaderSource };
    ShaderCode vertexShaderCode
      = new ShaderCode(GL3.GL_VERTEX_SHADER, sources.length, sources);
    boolean compiled = vertexShaderCode.compile(gl, System.err);
    if (!compiled)
      System.err.println("[error] Unable to compile vertex shader: " + sources);
    sources[0] = new String[]{ fragmentShaderSource };
    ShaderCode fragmentShaderCode
      = new ShaderCode(GL3.GL_FRAGMENT_SHADER, sources.length, sources);
    compiled = fragmentShaderCode.compile(gl, System.err);
    if (!compiled)
      System.err.println("[error] Unable to compile fragment shader: "
                        + sources);
    ShaderProgram program = new ShaderProgram();
    program.init(gl);
    program.add(vertexShaderCode);
    program.add(fragmentShaderCode);
    program.link(gl, System.out);
    if (!program.validateProgram(gl, System.out))
      System.err.println("[error] Unable to link program");
    return program.program();
  }

}
```

Program listing 6.1 continued: Shader.java