COM3503/4503/6503: 3D Computer Graphics

Dr Steve Maddock

Exercise sheet 3: Textures

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

1 Introduction

Last week we looked at shaders. A Shader class was introduced at the end of the exercise sheet to handle loading and compiling shaders. In exercise sheet 2, I mentioned that the Shader class could be extended to set unforms. I have done that and the result is supplied in the solutions to last week's exercise sheet on the website. You should take a look at that, as I will be using it this week.

This week we will mainly be looking at Textures. Essentially a 'texture' is a bitmap file (e.g. a jpg image) which can be used to colour a set of polygons. We'll look at the theory of this in lectures. For today, we'll look at the practicalities. I'll follow Joey's tutorial as I describe how to make the changes necessary for JOGL.

2 Texture coordinates

Program: Texture01.java. This first program loads a picture called 'wattBook.jpg' (Figure 2.1) and uses this to colour a single triangle. Figure 2.2 shows the triangle and the data for each vertex, which matches the data in Program listing 2.1. Figure 2.3 shows the result of running Texture01.

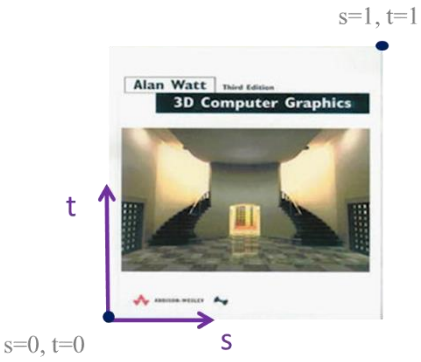

Figure 2.1: Filename 'wattBook.jpg' with overlaid texture coordinate system

Figure 2.3: Output from Texture01.java

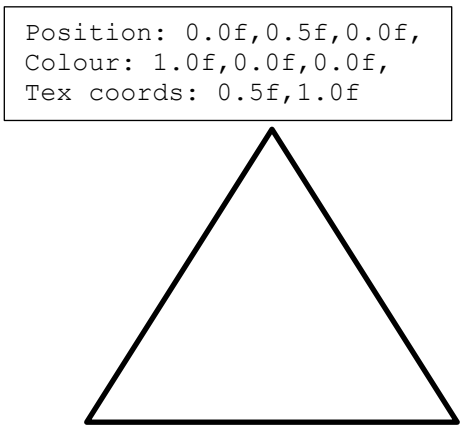

Figure 2.2: The data for a triangle: vertex position (x,y,z), vertex colour (r,g,b), texture coordinates (s,t)

```
private float[] vertices = {
    //  position (3 floats), colour (3 floats), tex cords (2 floats)
    0.0f,  0.5f, 0.0f,  1.0f, 0.0f, 0.0f,  0.5f, 1.0f,
    0.5f, -0.5f, 0.0f,  0.0f, 1.0f, 0.0f,  1.0f, 0.0f,
   -0.5f, -0.5f, 0.0f,  0.0f, 0.0f, 1.0f,  0.0f, 0.0f
};

private int vertexStride = 8;
private int vertexXYZFloats = 3;
private int vertexColourFloats = 3;
private int vertexTexFloats = 2;

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

Program listing 2.1: The data for the triangle. Each vertex has 3 attributes: vertex position (x,y,z), vertex colour (r,g,b), texture coordinates (s,t). In total, 8 floats are required per vertex.

Program listing 2.1 shows that each vertex now has 3 attributes, made up of 8 floats. The last two values for each vertex are the texture coordinates s and t, each in the range 0.0 to 1.0 inclusive. From Figure 2.3, you should be able to work out what is happening. Vertex 1 has texture coordinates 0.5, 1.0, which correspond to the top middle of the texture in Figure 2.1 – a texture has coordinates in the range [0, 1]. Vertex 2 has texture coordinates 1.0, 0.0, which corresponds to the bottom right of the texture image. Vertex 3 has texture coordinates 0.0, 0.0, which corresponds to the bottom left of the texture image. The texture coordinate values at the vertices are interpolated over the triangle surface during the rasterization process on the GPU (which occurs after the vertex shader and before the fragment shader). Using the fragment shader, each fragment that is produced then looks up the relevant coordinates in the texture image to retrieve the colour stored there – we call this sampling the texture (see Section 4).

An extra class variable, vertexTexFloats, has been introduced to describe that the texture coordinates are two floats. This variable is used in fillBuffers() in a similar way to previous exercise sheets. I'll leave you to look at the code – the main thing to note is that each vertex now has a position, a colour and texture coordinates, and the 'stride' and offset in the list of data have to be calculated accordingly, as illustrated in Joey's tutorial. It is worth noting that for our examples, the vertex attributes are interleaved (ABCABCABC). Thus, stride needs to step over consecutive sets of vertex attributes (ABC – in our examples this is 8 floats). For a data structure that is organised as AAABBBCCC (i.e. all positions, then all colours, then all texture coordinates) the stride would be calculated differently.

The vertex and fragment shaders need to be updated to deal with the extra texture coordinates data. First, let's look at how to load and create an OpenGL texture.

3 Loading and creating textures

Class: TextureLibrary.java. This class contains a static method to load an image from a file and create an OpenGL texture, as shown in Program listing 2.2. The method loadTexture() can be used to set some of the parameters described in Joey's tutorial. Make sure you read through Joey's description of the parameters that can be set for textures.

```
import java.io.File;
import java.io.FileInputStream;
import com.jogamp.opengl.*;
import com.jogamp.opengl.util.texture.spi.JPEGImage;

public final class TextureLibrary {
  // only deals with rgb jpg files

  public static int[] loadTexture(GL3 gl, String filename) {
    return loadTexture(gl, filename, GL.GL_REPEAT, GL.GL_REPEAT,
                                                   GL.GL_LINEAR, GL.GL_LINEAR);
  }

  public static int[] loadTexture(GL3 gl, String filename,
                          int wrappingS, int wrappingT, int filterS, int filterT) {
    int[] textureId = new int[1];
    try {
      File f = new File(filename);
      JPEGImage img = JPEGImage.read(new FileInputStream(f));
      gl.glGenTextures(1, textureId, 0);
      gl.glBindTexture(GL.GL_TEXTURE_2D, textureId[0]);
      gl.glTexParameteri(GL.GL_TEXTURE_2D, GL.GL_TEXTURE_WRAP_S, wrappingS);
      gl.glTexParameteri(GL.GL_TEXTURE_2D, GL.GL_TEXTURE_WRAP_T, wrappingT);
      gl.glTexParameteri(GL.GL_TEXTURE_2D, GL.GL_TEXTURE_MIN_FILTER, filterS);
      gl.glTexParameteri(GL.GL_TEXTURE_2D, GL.GL_TEXTURE_MAG_FILTER, filterT);
      gl.glTexImage2D(GL.GL_TEXTURE_2D, 0, GL.GL_RGB, img.getWidth(),
          img.getHeight(), 0, GL.GL_RGB, GL.GL_UNSIGNED_BYTE, img.getData());
      gl.glGenerateMipmap(GL.GL_TEXTURE_2D);
      /*gl.glTexParameteri(GL.GL_TEXTURE_2D, GL2.GL_TEXTURE_MIN_FILTER,
            GL2.GL_LINEAR_MIPMAP_LINEAR); */
      gl.glBindTexture(GL.GL_TEXTURE_2D, 0);
    }
    catch(Exception e) {
      System.out.println("Error loading texture " + filename);
    }
    return textureId;
  }
}
```

Program listing 3.1: TextureLibrary.java

Method initialise() in Texture01_GLEventListener.java calls loadTexture() and stores the result in the class variable textureId1. We'll look at the render method that uses this variable later.

```
  private int[] textureId1 = new int[1];

  public void initialise(GL3 gl) {
    shader = new Shader(gl, "t01_vs.txt", "t01_fs.txt");
    fillBuffers(gl);
    textureId1 = TextureLibrary.loadTexture(gl, "wattBook.jpg");
  }
```

Program listing 3.2: Method initialise() in Texture01_GLEventListener.java

4 The vertex and fragment shaders

Program listings 4.1 and 4.2 give the vertex and fragment shaders used in program Texture01. In Program listing 4.1, the vertex shader is altered to accept the texture coordinates for a vertex (in attribute location 2) and then forward the coordinates to the rasterization process and on to the fragment shader. In Program listing 4.2, the fragment shader accepts the texture coordinates from the rasterization process using 'ourTexCoord'. All that remains is to use the texture coordinates to look up a value in the texture and output a colour for the fragment. But how?

```
#version 330 core

layout (location = 0) in vec3 position;
layout (location = 1) in vec3 color;
layout (location = 2) in vec2 texCoord;

out vec3 ourColor;
out vec2 ourTexCoord;

void main() {
  gl_Position = vec4(position.x, position.y, position.z, 1.0);
  ourColor = color;
  ourTexCoord = texCoord;
}
```

Program listing 4.1: The vertex shader: t01_vs.txt

```
#version 330 core

in vec3 ourColor;
in vec2 ourTexCoord;

out vec4 color;

uniform sampler2D first_texture;

void main() {
  // color = vec4(ourColor, 1.0f);
  color = vec4(texture(first_texture, ourTexCoord).rgb, 1.0f);
  // color = vec4(texture(first_texture, ourTexCoord).rgb * ourColor, 1.0f);
}
```

Program listing 4.2: The fragment shader: t01_fs.txt

GLSL has a built-in data type for texture objects called a *sampler*. The postfix states what type of texture it is, in our case a sampler2D, i.e. a 2D texture. The texture is declared as a uniform since the same texture will be used for all instances of this fragment shader, i.e. the texture is the same constant for all the instances.

As it is a uniform, we need to pass it a value from the main application, as we have done for uniforms in programs on previous exercise sheets. We'll cover that in more detail soon.

Within the fragment shader, the GLSL function *texture()* is used. This samples the colour in the texture (first_texture) using the texture coordinates (ourTexCoord) and always returns a vec4 (i.e. an rgba colour), with component values in the range [0, 1].

It is worth briefly commenting on the code in the fragment shader. The texture function returns a value from the texture unit. Since the image we stored in this is a jpg image containing only rgb data, we need to be careful how we use this. Thus .rgb is used to access this data. We know that ourColor is a vec3. Thus the two sets of values can be multiplied together. The output colour (color) from the fragment shader is a vec4. We can construct this by using the result of the previous multiplication as the first three components of the vec4, followed by 1.0f as the fourth component.

Exercises

1. Program listing 4.2 contains three alternative ways to output a colour. Try uncommenting each line in turn to see what the effect is.
2. In the vertex shader, replace `ourTexCoord = texCoord;` with `ourTexCoord = texCoord + vec2(0.5f, -0.3f);`. Explain what is happening.
3. In Texture01_GLEventListener.java, method initialise(), replace

```
textureId1 = TextureLibrary.loadTexture(gl, "wattBook.jpg");
```

with

```
textureId1 = TextureLibrary.loadTexture(gl, "wattBook.jpg",
                          GL.GL_CLAMP_TO_EDGE, GL.GL_CLAMP_TO_EDGE,
                          GL.GL_LINEAR, GL.GL_LINEAR);
```

Again, can you explain what is happening?

5 Binding the texture

Focus on method render() in Texture01_GLEventListener.java – see Program listing 5.1.

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

  shader.use(gl);

  gl.glActiveTexture(GL.GL_TEXTURE0);
  gl.glBindTexture(GL.GL_TEXTURE_2D, textureId1[0]);

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

  // good practice to set everything back to defaults.
  gl.glActiveTexture(GL.GL_TEXTURE0);
}
```

Program listing 5.1: Method render() in Texture01_GLEventListener.java

Here, we state that GL_TEXTURE0 should be the active texture unit and we bind that to our texture using the variable textureId1 and a call to glBindtexture(). This is similar to binding the relevant vertex array before we draw a set of vertices.

So where was the link between the main program and the uniform declared in the fragment shader? There is an assumption that GL_TEXTURE0 is the default active texture unit and that if a sampler2D is used in the fragment shader it automatically refers to this. Thus, the uniform in the fragment shader was not explicitly set from the main program. Some would say this is sloppy programming, since not all drivers for GPUs will make this assumption. It could fail with some GPU drivers. To be safe, the uniform should be explicitly set by adding the following code:

```
gl.glUniform1i(gl.glGetUniformLocation(shader.getID(), "first_texture"), 0);
```

which is added to the render() method just after shader.use(gl). In fact, I have added a method to the class Shader, so this can instead be written as:

```
shader.setInt(gl, "first_texture", 0);
```

This code sets the uniform variable 'first_texture' in the fragment shader to refer to texture unit 0. We'll see the use of this in the next Section.

Exercises

1. In Texture01_GLEventListener.java, method initialise(), uses 'wattBook.jpg' as the filename. Try downloading a jpg image from the Internet and using this instead. Note, images with x and y dimensions that are both a power of 2 in size, e.g. 256x256, are preferred – see https://software.intel.com/en-us/articles/opengl-performance-tips-power-of-two-textures-have-better-performance for a discussion.
2. Program listing 2.1 gives the vertex data for the triangle. Change the texture coordinates for each vertex and predict the effect before running the program.

6 Texture Units

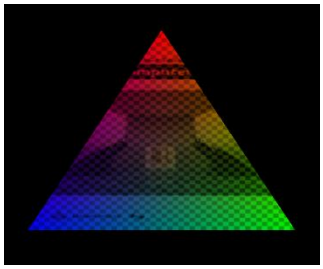

Figure 6.1: Output from Texture02.java

Program: Texture02.java

OpenGL has at least a minimum of 16 texture units: *GL_TEXTURE0* to *GL_TEXTURE15.* Program Texture02 uses two of these. The output is shown in Figure 6.1. each vertex is set to a different colour and two textures are mixed into this, 'wattBook.jpg' and 'chequerboard.jpg'. Program listing 6.1 gives the initialise() and render() methods in Texture02_GLEventListener.java.

Two textures are created and stored in variables textureId1 and textureId2, respectively. In render(), these are bound to texture units 0 and 1, which are associated with variables first_texture and second_texture in the fragment shader, which are shown in Program listings 6.2 and 6.3, respectively. However, it is inefficient to keep binding the same textures for every render call – see next section for a solution.

```
private int[] textureId1 = new int[1];
private int[] textureId2 = new int[1];

public void initialise(GL3 gl) {
  shader = new Shader(gl, "t02_vs.txt", "t02_fs.txt");
  fillBuffers(gl);
  textureId1 = TextureLibrary.loadTexture(gl, "wattBook.jpg");
  textureId2 = TextureLibrary.loadTexture(gl, "chequerboard.jpg");
}

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

  shader.use(gl);
  shader.setInt(gl, "first_texture", 0);
  shader.setInt(gl, "second_texture", 1);

  gl.glActiveTexture(GL.GL_TEXTURE0);
  gl.glBindTexture(GL.GL_TEXTURE_2D, textureId1[0]);
  gl.glActiveTexture(GL.GL_TEXTURE1);
  gl.glBindTexture(GL.GL_TEXTURE_2D, textureId2[0]);

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

  gl.glActiveTexture(GL.GL_TEXTURE0);
}
```

Program listing 6.1: Methods initialise() and render() in Texture02_GLEventListener.java

```
#version 330 core

layout (location = 0) in vec3 position;
layout (location = 1) in vec3 color;
layout (location = 2) in vec2 texCoord;

out vec3 ourColor;
out vec2 ourTexCoord;

void main() {
  gl_Position = vec4(position.x, position.y, position.z, 1.0);
  ourColor = color;
  ourTexCoord = texCoord;
}
```

Program listing 6.2: Vertex shader: t02_vs.txt

```
#version 330 core

in vec3 ourColor;
in vec2 ourTexCoord;

out vec4 color;

uniform sampler2D first_texture;
uniform sampler2D second_texture;

void main() {
  // color = vec4(ourColor, 1.0f);
  // color = vec4(texture(first_texture, ourTexCoord).rgb, 1.0f);
  // color = vec4(texture(first_texture, ourTexCoord).rgb * ourColor, 1.0f);
  // color = vec4(texture(second_texture, ourTexCoord).rgb * ourColor, 1.0f);
  color = vec4(mix(texture(first_texture, ourTexCoord),
                   texture(second_texture, ourTexCoord), 0.3f).rgb
               * ourColor, 1.0f);
}
```

Program listing 6.3: Fragment shader: t02_fs.txt

The fragment shader in Program listing 6.3 is the interesting one. The two textures are combined using the GLSL function called *mix*[1], which takes two colours and a floating point value as parameters. A value of 0.3 returns a mixture of 70% of the first colour and 30% of the second colour. The result of the mix function is then multiplied by the input colour of the fragment (which was output by the rasterization stage which interpolated values from the vertices of the respective triangle).

Exercises

1. Try some of the other commented out lines in the fragment shader. Predict the effect each time before you run the program.
2. Add a uniform variable that varies the amount of mix over time.
3. Add a third texture into the mix.

[1] http://docs.gl/sl4/mix

7 Texture binding and efficiency

Program: Texture03. java

This program is nearly identical to Texture02.java. The only change is shown in Program listing 7.1. The calls to make a texture unit active and to bind a particular texture to the texture unit are done in the method initialise() after loading the textures form file. Method render() now only assigns the name of the relevant uniform in the fragment shader to each texture unit. This is more efficient than continually rebinding the texture units.

We could load a collection of textures in this way and store them in a library of textures associated with texture units. Then in the main render loop, we would just assign which texture unit we wished a particular uniform sampler2D to refer to. Of course, if we have more textures than texture units, then we will need to do some rebinding and switching of active textures in the render loop. However, we shouldn't need more than 16 textures in the examples for the exercise sheets.

```
private int[] textureId1 = new int[1];
private int[] textureId2 = new int[1];

public void initialise(GL3 gl) {
  shader = new Shader(gl, "t02_vs.txt", "t02_fs.txt");
  fillBuffers(gl);
  textureId1 = TextureLibrary.loadTexture(gl, "wattBook.jpg");
  textureId2 = TextureLibrary.loadTexture(gl, "chequerboard.jpg");
  // bind the two textures to the first two texture units
  gl.glActiveTexture(GL.GL_TEXTURE0);
  gl.glBindTexture(GL.GL_TEXTURE_2D, textureId1[0]);
  gl.glActiveTexture(GL.GL_TEXTURE1);
  gl.glBindTexture(GL.GL_TEXTURE_2D, textureId2[0]);
}

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

  shader.use(gl);
  shader.setInt(gl, "first_texture", 0);
  shader.setInt(gl, "second_texture", 1);

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

Program listing 7.1: Methods initialise() and render() in Texture03_GLEventListener.java

There are other efficiencies that could be considered in our programs. For example, it is relatively slow to get the location of a uniform from the GPU. Instead, the locations of uniforms could be queried in the initialise method and stored in global variables. Then, in the render method they would only need setting to particular values. Also, if a uniform does not change from one render call to the next, there is no need to keep resetting its value. The uniform will retain its value once set in the shader program – it is deemed to be *active* until the shader program is relinked.

We'll not worry about these extra efficiencies and just use the methods we have written in the Shader class to set the values of named uniforms from within the render loop. This makes the code clearer whilst we are learning about OpenGL.

8 Dynamic textures

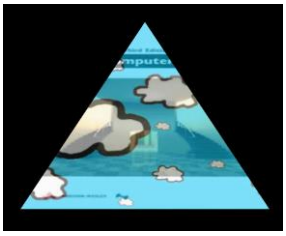

Figure 8.1: Output from Texture04.java

Program: Texture04.java. Two textures are used in this program. One texture is static, whilst the other texture continually scrolls over the top of the other texture (see Figure 8.1).

This program uses a uniform in the vertex shader to vary the texture coordinates over time for one of the textures. Program listing 8.1 gives the render() method. Program listings 8.2 and 8.3 give the vertex and fragment shaders, respectively.

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

  double elapsedTime = getSeconds() - startTime;

  shader.use(gl);

  double t = elapsedTime*0.1;
  float offsetX = (float)(t - Math.floor(t));
  float offsetY = 0.0f;
  shader.setFloat(gl, "offset", offsetX, offsetY);

  shader.setInt(gl, "first_texture", 0);
  shader.setInt(gl, "second_texture", 1);

  gl.glActiveTexture(GL.GL_TEXTURE0);
  gl.glBindTexture(GL.GL_TEXTURE_2D, textureId1[0]);
  gl.glActiveTexture(GL.GL_TEXTURE1);
  gl.glBindTexture(GL.GL_TEXTURE_2D, textureId2[0]);

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

  gl.glActiveTexture(GL.GL_TEXTURE0);
}
```

Program listing 8.1: Method render() in Texture04_GLEventListener.java

The uniform called offset in the vertex shader is updated each time render() is called. This value is added to the texCoord to produce movingTexCoord which is then passed to the rasterizer, which interpolates the values from each vertex over a triangle, producing a set of fragments which each have a colour (ourColor) and two sets of texture coordinates (ourTexCoord and movingTexCoord). ourTexCoord is used in the fragment shader to sample first_texture and movingTexCoord is used in the fragment shader to sample second_texture. The results are mixed together using the GLSL mix function.

An alternative solution would place the uniform offset in the fragment shader instead of the vertex shader. The offset would then be added to the interpolated ourTexCoord value.

Exercises

1. Change the calculation of offset in method render() to see what dynamic effect you can create.
2. If you are feeling adventurous, find some different textures to use to produce a different dynamic texture effect.

```
#version 330 core

layout (location = 0) in vec3 position;
layout (location = 1) in vec3 color;
layout (location = 2) in vec2 texCoord;

out vec3 ourColor;
out vec2 ourTexCoord;
out vec2 movingTexCoord;

uniform vec2 offset;

void main() {
  gl_Position = vec4(position.x, position.y, position.z, 1.0);
  ourColor = color;
  ourTexCoord = texCoord;
  movingTexCoord = texCoord + offset;
}
```

Program listing 8.2: Vertex shader: t04_vs.txt

```
#version 330 core

in vec3 ourColor;  // not needed
in vec2 ourTexCoord;
in vec2 movingTexCoord;

out vec4 color;

uniform sampler2D first_texture;
uniform sampler2D second_texture;

void main() {
  color = vec4(mix(texture(first_texture, ourTexCoord),
                   texture(second_texture, movingTexCoord),
                   0.5f).rgb, 1.0f);
}
```

Program listing 8.3: Fragment shader: t04_fs.txt

9 Changing textures

Program: Texture05.java. The final program loads 4 textures into memory and then displays each of these in turn on the same triangle. Program listing 9.1 gives the initialise() and render() methods. Program listings 9.2 and 9.3 give the vertex and fragment shaders, respectively.

```
private int[] textureId1 = new int[1];
private int[] textureId2 = new int[1];
private int[] textureId3 = new int[1];
private int[] textureId4 = new int[1];
private final int NUM_TEXTURES = 4;

public void initialise(GL3 gl) {
  shader = new Shader(gl, "t05_vs.txt", "t05_fs.txt");
  fillBuffers(gl);
  textureId1 = TextureLibrary.loadTexture(gl, "paintstrokes/wattBook.jpg");
  textureId2 = TextureLibrary.loadTexture(gl, "paintstrokes/wattBook1.jpg");
  textureId3 = TextureLibrary.loadTexture(gl, "paintstrokes/wattBook2.jpg");
  textureId4 = TextureLibrary.loadTexture(gl, "paintstrokes/wattBook3.jpg");
  gl.glActiveTexture(GL.GL_TEXTURE0);
  gl.glBindTexture(GL.GL_TEXTURE_2D, textureId1[0]);
  gl.glActiveTexture(GL.GL_TEXTURE1);
  gl.glBindTexture(GL.GL_TEXTURE_2D, textureId2[0]);
  gl.glActiveTexture(GL.GL_TEXTURE2);
  gl.glBindTexture(GL.GL_TEXTURE_2D, textureId3[0]);
  gl.glActiveTexture(GL.GL_TEXTURE3);
  gl.glBindTexture(GL.GL_TEXTURE_2D, textureId4[0]);
  startTime = getSeconds();
}

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

  double elapsedTime = getSeconds() - startTime;
  int currentTexture
           = (int)Math.floor(2*(1+Math.sin(elapsedTime+Math.toRadians(270))));
  shader.use(gl);
  shader.setInt(gl, "first_texture", currentTexture);

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

Program listing 9.1: Methods initialise() and render() in Texture05_GLEventListener.java

```
#version 330 core

layout (location = 0) in vec3 position;
layout (location = 1) in vec3 color;
layout (location = 2) in vec2 texCoord;

out vec3 ourColor;
out vec2 ourTexCoord;

void main() {
  gl_Position = vec4(position.x, position.y, position.z, 1.0);
  ourColor = color;
  ourTexCoord = texCoord;
}
```

Program listing 9.2: Vertex shader: t05_vs.txt

```
#version 330 core

in vec3 ourColor;
in vec2 ourTexCoord;

out vec4 color;

uniform sampler2D first_texture;

void main() {
  color = vec4(texture(first_texture, ourTexCoord).rgb, 1.0f);
}
```

Program listing 9.3: Fragment shader: t05_fs.txt