# COM3503/4503/6503: 3D Computer Graphics

Dr Steve Maddock

# Exercise sheet 1: Introduction to JOGL and modern OpenGL

**Before you start this exercise sheet,** set up your JOGL environment as described on the module web site in the 'starting JOGL' document.

It's now time to write your first programs in JOGL. I will assume that a text editor is being used to create programs which are then compiled and run from a Windows command tool. The sample programs (S01.java. S02.java and S03.java) can be downloaded from the module Web site.

## 1 Creating a window

### 1.1 Importing Java packages

Some of you will have programmed with Java to the level of creating windows with either the AWT or with Swing, or both. We'll use Swing in the following examples.

Program Listing 1.1-1.3 (S01.java) opens a window and attaches an OpenGL canvas and event listener. Nothing is drawn on the canvas area at this stage. However, this program can be used to check that you have set up the relevant paths for JOGL and thus the program compiles. We'll use this program to explain some of the housekeeping associated with setting up a window – this code will appear in many of your future programs.

I have chosen to print out the program code here so that you have something to refer to at a later date. When you examine the online code, you will see that I have also added some comments to explain some of the JOGL.

| | |
|---|---|
| 1 | `import java.awt.BorderLayout;`<br>`import java.awt.Dimension;`<br>`import java.awt.event.WindowAdapter;`<br>`import java.awt.event.WindowEvent;`<br>`import javax.swing.JFrame;`<br>`import com.jogamp.opengl.GLAutoDrawable;`<br>`import com.jogamp.opengl.GLCapabilities;`<br>`import com.jogamp.opengl.GLProfile;`<br>`import com.jogamp.opengl.GLEventListener;`<br>`import com.jogamp.opengl.awt.GLCanvas;`<br>`import com.jogamp.opengl.util.FPSAnimator;` |
| 2 | `public class S01 extends JFrame {` |

**Program listing 1.1:** Part of S01.java: Importing packages

First, a number of packages need to be imported (see Program listing 11., line 1 – rather than use lots of line numbers I've put all the lines in one cell in the table and called it line 1). These give access to the OpenGL API and also the standard Java routines for creating a window and detecting events on that window (e.g. the window closing event, triggered in Microsoft Windows by clicking on the cross at the right end of the window title bar). If you don't know how to use Java AWT and Java Swing, then you will need to do some background reading, as I will be focussing on the OpenGL aspects. For now, you can just reuse the same code in each of your programs.

I have separately listed individual classes in the import statements. An alternative is to use java.awt.* and java.awt.event.* and com.jogamp.opengl.* and com.jogamp.opengl.util.* and so on. The compiler will then decide which classes need to be imported.

To create the window, we use Java Swing's JFrame class. The main class S01 extends the JFrame class, so the code to create the frame is inherited when the constructor is called (see Program listing 1.2).

```
3   public class S01 extends JFrame {
4     private static final int WIDTH = 1024;
      private static final int HEIGHT = 768;
      private static final Dimension dimension = new Dimension(WIDTH, HEIGHT);
5     private GLCanvas canvas;
      private GLEventListener glEventListener;
6     private final FPSAnimator animator;
7
8     public static void main(String[] args) {
9       S01 f = new S01("S01");
10      f.getContentPane().setPreferredSize(dimension);
11      f.pack();
12      f.setVisible(true);
13    }
14
15    public S01(String textForTitleBar) {
16      // ...
```

**Program listing 1.2:** Part of S01.java: Creating a Frame

Three private attributes are declared in the S01 class. One is static and is the dimensions of the window that will be created. One is the canvas which will be drawn on. The third will be used to create a display loop, i.e. a loop that is executed n times per second to display whatever needs to be drawn (see more detail later). The main entry point to the class creates an instance of the S01 class, i.e. a frame, sets its size, uses pack to make sure that all the contents of the frame are at or above their preferred sizes, and then makes the frame visible.

The S01 constructor is given in Program listing 1.3. This sets certain parameters and adds widgets and listeners to the frame (window). Line 18 sets the text displayed in the title bar using the superclass. Before explaining the remaining lines, we need to learn a little more about profiles.

```
17    public S01(String textForTitleBar) {
18      super(textForTitleBar);
19      GLCapabilities glcapabilities
          = new GLCapabilities(GLProfile.get(GLProfile.GL3));
20      canvas = new GLCanvas(glcapabilities);
21      glEventListener = new S01_GLEventListener();
22      canvas.addGLEventListener(glEventListener);
23      getContentPane().add(canvas, BorderLayout.CENTER);
24      addWindowListener(new WindowAdapter() {
25        public void windowClosing(WindowEvent e) {
26          animator.stop();
27          remove(canvas);
28          dispose();
29          System.exit(0);
30        }
31      });
32      animator = new FPSAnimator(canvas, 60);
33      animator.start();
34    }
35
36  }
```

**Program listing 1.3:** The S01 constructor

## 1.2 OpenGL profiles

OpenGL continues to evolve, with new versions adding functionality and even deprecating functionality. New functionality may only be available on certain hardware, and it must be possible to continue to use existing hardware. Thus, a programmer must be given control over which version of OpenGL to use. This is done using GLProfile. A programmer can choose to set a particular OpenGL version. It is even possible to write code for different versions of OpenGL and choose which code to execute at run time. In JOGL, the following code would select the desktop OpenGL core profile 3.x, for x>=1: GLProfile.get(GLProfile.GL3).

| **OpenGL versions** | |
|---|---|
| 1.0-1.5 | Fixed function pipeline |
| 2.0-2.1 | Support for programmable shaders |
| 3.0 | Adopts deprecation model (fixed function deprecated), but retains backward compatibility |
| 3.x | Fixed function pipeline and associated functions removed (but can be accessed using a compatibility context) |
| 4.x | Geometry and tessellation shaders |
| ES 1.x | Fixed-function version (stripped down) [ES - embedded systems] |
| ES 2.x | Shader-only version (for mobile |
| ES 3.x | Geometry and tessellation shaders |

**Table 1.1:** OpenGL versions

## 1.3 A drawing surface

Line 19 of the program (in Program listing 1.3) specifies the capabilities of the OpenGL rendering context[1]. We're initialising these with the OpenGL 3.x profile. Line 20 creates the GLCanvas on which all OpenGL drawing will occur, and this is added to the centre of the frame on line 23 – we'll come back to lines 21 and 22 in a short while.

(Note: A GLCanvas is an implementation of the interface GLAutoDrawable which is a subinterface of GLDrawable, which is the abstraction of the OpenGL rendering target. The GLDrawable or GLAutoDrawable maintains the rendering context. In other words, the GLCanvas, which is the implementation, maintains the context. If a GLAutoDrawable is used the context is automatically created. We'll use a GLAutoDrawable in our programs – see S01_GLEvenetListener.java in the next section).

Summarising: 1. Choose GL Profile; 2. Configure GLCapabilities; 3. Create GLCanvas (implementation of GLAutoDrawable), which automatically creates the OpenGL context; 4. Add the GLCanvas to the JFrame (i.e. the window).

## 1.4 GLEventListener

Lines 21 and 22 are important lines:

glEventListener = new S01_GLEventListener();

canvas.addGLEventListener(glEventListener);

These lines create a new instance of the class S01_GLEventListener and add it to the canvas. Practically, this means that the canvas will respond to four particular GL events. The four events that it will respond to are (i) to initialise the OpenGL context; (ii) to do something when the window is

[1] From http://jogamp.org/deployment/jogamp-next/javadoc/jogl/javadoc/: class GLContext: "In order to perform OpenGL rendering, a context must be "made current" on the current thread. OpenGL rendering semantics specify that only one context may be current on the current thread at any given time, and also that a given context may be current on only one thread at any given time. Because components can be added to and removed from the component hierarchy at any time, it is possible that the underlying OpenGL context may need to be destroyed and recreated multiple times over the lifetime of a given component. This process is handled by the implementation, and the GLContext abstraction provides a stable object which clients can use to refer to a given context."

resized by the user; (iii) to display the result of some OpenGL commands on the canvas – the display event can be triggered in a number of ways – see animation below; (iv) to dispose of any memory that has been used whilst dealing with OpenGL on the GPU. Further details about the class S01_GLEventListener are given below.

## 1.5 A window listener

Lines 24-31 of Program Listing 1c add a window listener to the frame using an anonymous inner class (which is an advanced feature of Java[2]). Suffice to say, this handles the closing of the window when, in Microsoft Windows, the user selects the cross at the right of the title bar for a window. Some clean-up operations are also carried out. We'll skip over the details of this code, and accept it as 'boilerplate' code.

## 1.6 Animation

Lines 32 and 33 in Program listing 1.3 create a JOGL FPSAnimator instance and attach it to the canvas. The parameters specify that the canvas should redraw at 60 frames per second (i.e. a display method is called 60 times per second). (Here, I use the word 'frame' to signify one scene rendering in an animation loop and this should not be confused with the use of the Frame for the window.) The FPSAnimator is part of the package `com.jogamp.opengl.util.*` which is why it is imported at the start of the program on line 6.

## 1.7 A rendering loop

At this point we are now ready to write the rendering code for the program. Figure 1.1 summarises the render loop for a real-time rendering process. The alternative is offline rendering where the rendering of a single image can take minutes, hours or even days. For real-time rendering, input from the user, the simulation update process and the rendering of the data to create an image must all finish within the allotted time. So, if an FPSAnimator is set at 60 frames per second, then the update and render must take less than approximately 17ms. (It is this fight against time that leads to research into techniques to simulate and render efficiently.)

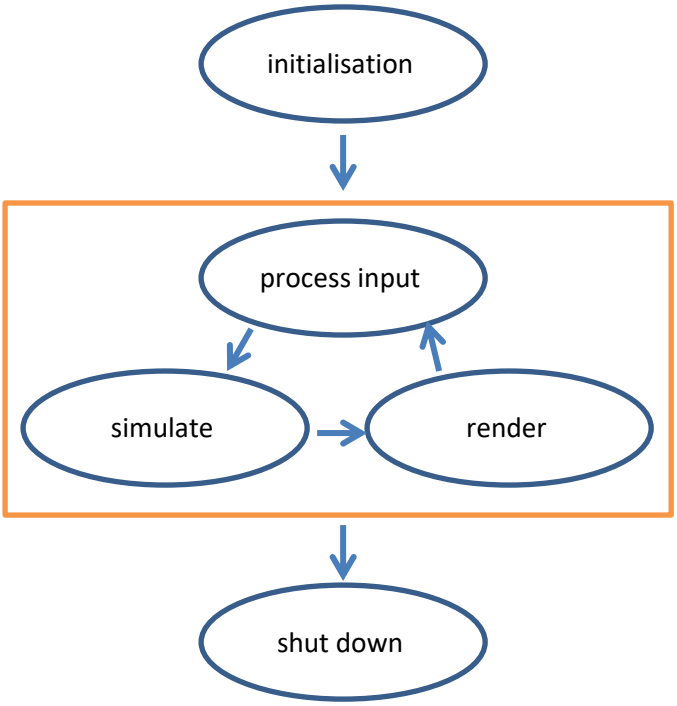


**Figure 1.1:** Summary of rendering loop

In Figure 2, 'process input' deals with user input, e.g. mouse or keyboard presses in the interface, perhaps to change the position of the virtual camera that is being used to view the 3D world. 'Simulate' means that the 3D world is updated in some way. This could be applying physics, checking for collisions between objects, updating object geometry, etc. Finally, 'render' draws the objects, producing an on-screen image for the particular camera viewpoint.

## 1.8 Rendering a blank screen!!

Program listing 1.4 gives the details of S01_GLEventListener.java. Four of its methods must be included – init, reshape, display and dispose – because it implements the GLEventListener interface (line 3), which specifies that these methods must be provided. The constructor for the class is empty – we don't need it for anything at this stage. It is there because I chose to implement the

[2] http://docs.oracle.com/javase/tutorial/java/javaOO/innerclasses.html

GLEventListener interface as a separate class, S01_GLEventListener. An alternative would have been for the class S01 to implement GLEventListener, with the methods init, reshape, display and dispose defined in S01.java. However, I felt that it was better to separate the code for handling GL events into a separate class to reduce the clutter of code in S01.java. Thus, we have a constructor that does nothing in S01_GLEventListener.

Method init is used, as its name suggest, to set up certain attributes in the OpenGL context. The parameter drawable is automatically supplied by the system. When the GLEventListener is created, the init method is automatically called. We thus use it to set up the data structures we will require for the OpenGL work and also to set up relevant OpenGL attributes. Here, I have included more than is necessary. Line 9 is used to get the OpenGL object from drawable. I have indicated here that I want an OpenGL3 version of this. Lines 10 to 14 are unnecessary in most programs. I have included them here so that you will see some of the capabilities of the system that you are running on.

```
1   import com.jogamp.opengl.*;
    import com.jogamp.opengl.util.*;
    import com.jogamp.opengl.util.awt.*;
2
3   public class S01_GLEventListener implements GLEventListener {
4
5     public S01_GLEventListener() {
6     }
7
8     public void init(GLAutoDrawable drawable) {
9       GL3 gl = drawable.getGL().getGL3();
10      System.err.println("Chosen GLCapabilities: "
          + drawable.getChosenGLCapabilities());
11      System.err.println("INIT GL IS: " + gl.getClass().getName());
12      System.err.println("GL_VENDOR: " + gl.glGetString(GL.GL_VENDOR));
13      System.err.println("GL_RENDERER: " + gl.glGetString(GL.GL_RENDERER));
14      System.err.println("GL_VERSION: " + gl.glGetString(GL.GL_VERSION));
15      gl.glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
16      gl.glClearDepth(1.0f);
17      gl.glEnable(GL.GL_DEPTH_TEST);
18      gl.glDepthFunc(GL.GL_LESS);
19    }
20
21    public void reshape(GLAutoDrawable drawable, int x, int y,
                          int width, int height) {
        GL3 gl = drawable.getGL().getGL3();
        gl.glViewport(x, y, width, height);
22    }
23
24    public void display(GLAutoDrawable drawable) {
25      GL3 gl = drawable.getGL().getGL3();
26      render(gl);
27    }
28
29    public void dispose(GLAutoDrawable drawable) {
30    }
31
32    public void render(GL3 gl) {
33      gl.glClear(GL.GL_COLOR_BUFFER_BIT | GL.GL_DEPTH_BUFFER_BIT);
34    }
35
36  }
```

**Program listing 1.4:** S01_GLEventListener.java

Line 15 uses the OpenGL command glClearColor(r,g,b,alpha) to set the OpenGL attribute that controls what colour (note that the US version 'color' is used in the command) the canvas area is when it is cleared. Here it is set to black. The last parameter is usually set to 1.0f (where the f signifies that it is a float value, not a double – be careful in Java, as it is very picky about supplying the right type as a parameter for many of the JOGL methods). (Note: In C, the command would just be glClearColor(). In Java, it is gl.glClearColor(), because of the way that JOGL is implemented to fit in with Java's ethos. This can cause some confusion when you read online programs, which are inevitably written in C.)

Lines 16, 17 and 18 relate to something called the z buffer which is used when rendering objects that have depth, i.e. are set at different distances away from the camera system – we will deal with these in detail when we look at camera systems. In brief, the z buffer is used for hidden surface removal, where one surface should appear in front of another surface, or surface part in the case of intersecting surfaces.

The reshape function and the dispose function are both left empty at this stage. The reshape function is automatically called by the system in respond to the user resizing the window, i.e. producing a window resize event. The dispose function is called when the GLEventListener instance is deleted.

The display function is where we draw something. This function will be called whenever the window needs redrawing as the result of a user event or an animation loop, such as the one using FPSAnimator in Program listing 1.3. Again, the OpenGL object is retrieved. This is then passed to the render method. There is no good reason for having a separate render method at this stage. The code in it could just as easily be inside the display method. However, it will be useful to have it separate in later more complex programs, so I have also separated it here.

The render method does two things: it clears the screen to the clear colour we set up in the init method, and it also clears the depth buffer (i.e. the z buffer which will be used for hidden surface removal in later programs).

We have now examined a lot of code in order to display a blank screen!! However, it has been useful, since much of the code for setting up a window will be identical in subsequent programs. These programs will focus on the OpenGL aspects and will concentrate on the methods in the GLEventListener instance.

### Exercises

1. Line 4 in program listing 1.2 can be used to alter the initial size of the window. Try changing the two values here.
2. Alter the background colour of the screen.

# 2 Drawing a triangle

## 2.1 Introduction

The next step is to draw a single triangle. In modern OpenGL, this is more detailed than you might expect, as there is a lot of work to be done on setting up buffers and shaders. Figure 2.1 illustrates what has to be done. The vertex and fragment shaders have to be supplied and compiled. Buffers on the GPU have to be initialised with the data that describes the triangle. Then, the display loop can start and use the shader programs to render the data stored in the buffers.

## 2.2 Learn OpenGL

I recommend that you read through Joey de Vries's Learn OpenGL tutorial https://learnopengl.com. It is an excellent introduction to modern OpenGL, although it is written for C rather than Java. Nonetheless the main OpenGL ideas are relevant and can be transferred. In the practical sessions for this module, we will cover the material that is included in the sections 'Getting started', 'Lighting' and 'Model Loading'. For my module, you do not need to look at the more advanced OpenGL material that he covers. I will continue to refer to this tutorial in subsequent weeks.

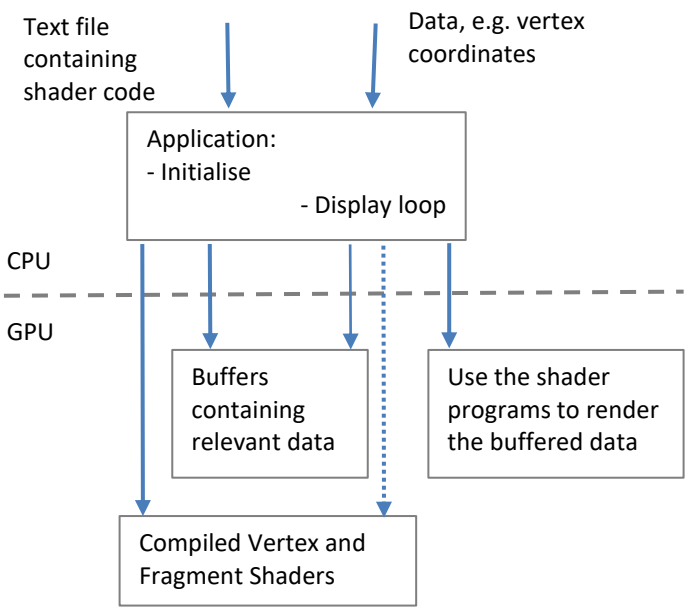


**Figure 2.1:** Setting up buffers and shaders

## 2.3 The triangle

The full program is long and contains a lot of details that need explaining, so I have split this into a series of pieces (Program listings 2.1-2.5). The program is similar to Joey de Vries's example at https://learnopengl.com/#!Getting-started/Hello-Triangle and I urge you to read through that. I will not explain the OpenGL details, since he gives those. Instead, I'll mainly focus on the JOGL differences, since some of the JOGL methods have different parameters, and, of course, Java does not have pointers, which some of his C examples depend on. (It is sometimes useful to look up the JOGL documentation at: https://jogamp.org/deployment/webstart/javadoc/jogl/javadoc/.)

Program listing 2.1 is similar to Program listing 1.4. The main difference is the call to initialise(gl) in method init() and some housekeeping in method dispose() to clean up memory on the GPU. There are also extra import statements since we will need some of Java's more esoteric classes to deal with transfer of data to the GPU. There are four sections at the end of Program listing 2.1 that indicate where the code from Program listings 2.2-2.5 goes. The first section is the 'scene', by which I mean the methods to initialise the scene and to render it. The second section, 'data', is where the triangle data will be declared. The third section, 'buffers', is where the buffers are set up on the GPU and filled with data. The fourth section, 'shaders', is for setting up the vertex and fragment shaders on the GPU. One of the reasons for using these different labelled sections is that it hints at the structure of later, more complex programs, where some of the sections will become separate classes, e.g. the data section. For now, we'll keep everything in the one class so that it is easier to see what is happening 'as a whole'.

Let's start with the data section. Program listing 2.2 gives the vertex data for a single triangle, defined as a list of nine floats, which are the x,y,z coordinates for each of three vertices. At this stage, our triangle has no depth and so all the z values are 0. In later examples, other data will be stored for vertices and a separate data structure will also be used to describe how to join the vertex data together to make a mesh of triangles. A single triangle suits us for this first example.

Program listing 2.3 sets up the buffers on the GPU. I'll focus on the differences to Joey de Vries's example. The C versions of glGenVertexArrays() and glGenBuffers() both return an address in GPU memory. The JOGL equivalents handle this by using an array and storing the address in the first array location. The first parameter in the methods states how many buffers are required, which is why an array is used, so that multiple addresses can be returned. We only need one buffer for the vertex array object and one buffer for the vertex buffer object. The variables vertexBufferId and vertexArrayId are declared as attributes of the class, since they may be required in other methods in

the class. The JOGL version of glBufferData() is different to its C cousin. The third parameter in the JOGL version is a java.nio.FloatBuffer. The vertex x,y,z data has to be transferred into a variable of this type before it can be passed to glBufferData(). The rest of the method fillBuffers() should be clear from following Joey de Vries's example.

Program listing 2.4 sets up and compiles the shaders on the GPU. At this stage, the code for the shaders is given as two Strings in the program. Method initialiseShader() just displays the Strings. In later examples, the shaders will be loaded from files. The variable shaderProgram is declared as an attribute of the class so that it can be referred to in other methods of the class. The method compileAndLink() can be done in a number of ways. Program Listing 2.4 uses JOGL helper classes to achieve it. Program listing 2.5 is an alternative way to achieve it that is similar to Joey de Vries's example. (There is also some extra code which is concerned with error checking the shaders.) Program listing 2.4 is much better as it gives easier access to error checking and the code is more compact. The only change we'll make to this code in later examples is to load the vertex and fragment shader text from files. Otherwise, the code will remain as is, so we can treat it as 'boilerplate' code.

Program listing 2.6 gives the methods to initialise and render the 'scene' (the single triangle!!). The first method, initialise() ,uses the methods described above to setup the shaders and buffers. The second method, render(), begins by clearing the screen and depth buffers. The particular shader program to use is then set. We can see now why there is an attribute called shaderProgram. Multiple shader programs could be created with different vertex and fragment shaders, and different parts of the scene could be rendered with different shaders. Then the relevant vertex data on the GPU is selected. Again, we could store multiple lists of vertex data on the GPU and choose which one we wished to render. The command glDrawArrays() is issued so that the GPU renders the data in the chosen buffer with the chosen shader program. The parameters state that a triangle is to be drawn, that the data starts at index position 0 in the buffer and that it is 3 vertices long. (For multiple triangles, the value 3 would change accordingly, depending on how the vertex data is stored in the buffer, e.g. if data is repeated at shared vertices or not – see later examples.) Finally, we unbind the vertex array. At this point a different vertex array and shader program could be bound and a different set of data displayed.

## Exercises

1. Change the colour of the triangle that is drawn. (Hint: change the fragment shader.) Examples on future exercise sheets will show how this can be done from the application rather that by hard-coding it into the fragment shader.
2. Without using Element Buffer Objects (see next section), try to change the program to draw two triangles. (Hint: define a second triangle in the 'data' section, by extending the existing array of vertices, then, in method render() in the 'scene' section, change gl.glDrawArrays(GL.GL_TRIANGLES, 0, 3); accordingly.

```
import java.nio.*;
import com.jogamp.common.nio.*;
import com.jogamp.opengl.*;
import com.jogamp.opengl.util.*;
import com.jogamp.opengl.util.awt.*;

public class S02_GLEventListener implements GLEventListener {

  public S02_GLEventListener() {
  }

  public void init(GLAutoDrawable drawable) {
    GL3 gl = drawable.getGL().getGL3();
    gl.glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
    gl.glClearDepth(1.0f);
    gl.glEnable(GL.GL_DEPTH_TEST);
    gl.glDepthFunc(GL.GL_LESS);
    initialise(gl);
  }

  public void reshape(GLAutoDrawable drawable, int x, int y, int width,
int height) {
  }

  public void display(GLAutoDrawable drawable) {
    GL3 gl = drawable.getGL().getGL3();
    render(gl);
  }

  public void dispose(GLAutoDrawable drawable) {
    GL3 gl = drawable.getGL().getGL3();
    gl.glDeleteBuffers(1, vertexBufferId, 0);
  }


  // ***************************************************
  /* THE SCENE */


  // ***************************************************
  /* THE DATA */


  // ***************************************************
  /* THE BUFFERS */


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

}
```

**Program listing 2.1:** `S02_GLEventListener.java`

```
  // ***************************************************
  /* THE DATA
   */

  // one triangle
  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
  };
```

**Program listing 2.2:** The data – a single

```
// ***************************************************
  /* THE BUFFERS
   */

  private int[] vertexBufferId = new int[1];
  private int[] vertexArrayId = new int[1];

  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);

    gl.glVertexAttribPointer(0, 3, GL.GL_FLOAT, false, 3*Float.BYTES, 0);
    gl.glEnableVertexAttribArray(0);

    gl.glBindBuffer(GL.GL_ARRAY_BUFFER, 0);
    gl.glBindVertexArray(0);
  }
```

**Program listing 2.3:** Setting up buffers

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

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, 1.0f);\n" +
  "}";

private String fragmentShaderSource =
  "#version 330 core\n" +
  "\n" +
  "out vec4 color;\n" +
  "\n" +
  "void main() {\n" +
  "  color = vec4(0.1f, 0.7f, 0.9f, 1.0f);\n" +
  "}";

private int shaderProgram;

private void initialiseShader(GL3 gl) {
  System.out.println(vertexShaderSource);
  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 2.4:** Setting up the shaders – JOGL provides helper classes to create a shader program

```
private int alternative_compileAndLink(GL3 gl) {
  int vertexShader = gl.glCreateShader(GL3.GL_VERTEX_SHADER);
  String[] source = new String[]{ vertexShaderSource };
  int[] sLength = new int[1];
  sLength[0] = source[0].length();
  gl.glShaderSource(vertexShader, 1, source, sLength, 0);
  gl.glCompileShader(vertexShader);
  if (shaderError(gl, vertexShader, "Vertex shader"))
    System.exit(0);

  int fragmentShader = gl.glCreateShader(GL3.GL_FRAGMENT_SHADER);
  source = new String[]{fragmentShaderSource};
  sLength[0] = source[0].length();
  gl.glShaderSource(fragmentShader, 1, source, sLength, 0);
  gl.glCompileShader(fragmentShader);
  if (shaderError(gl, fragmentShader, "Fragment shader"))
    System.exit(0);

  int shaderProgram = gl.glCreateProgram();
  gl.glAttachShader(shaderProgram, vertexShader);
  gl.glAttachShader(shaderProgram, fragmentShader);
  gl.glLinkProgram(shaderProgram);
  gl.glValidateProgram(shaderProgram);

  gl.glDetachShader(shaderProgram,vertexShader);
  gl.glDetachShader(shaderProgram,fragmentShader);

  return shaderProgram;
}

// ************************************************
/* ERROR CHECKING for shader compiling and linking.
 */

private boolean shaderError(GL3 gl, int obj, String s) {
  int[] params = new int[1];
  gl.glGetShaderiv(obj, GL3.GL_COMPILE_STATUS, params, 0);
  boolean error = (params[0] == GL.GL_FALSE);
  if (error) {
    gl.glGetShaderiv(obj, GL3.GL_INFO_LOG_LENGTH, params, 0);
    int logLen = params[0];
    byte[] bytes = new byte[logLen + 1];
    gl.glGetShaderInfoLog(obj, logLen, null, 0, bytes, 0);
    String logMessage = new String(bytes);
    System.out.println("\n***ERROR***");
    System.out.println(s + ": "+logMessage);
  }
  return error;
}
```

**Program listing 2.5:** Alternative version of compile and link for setting up the shaders. This is closer in style to Joey de Vries's example, but we won't use this version.

```
// ************************************************
/* THE SCENE
 * Now define all the methods to handle the scene.
 * This will be added to in later examples.
 */

public void initialise(GL3 gl) {
  initialiseShader(gl);
  shaderProgram = compileAndLink(gl);
  fillBuffers(gl);
}

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.glDrawArrays(GL.GL_TRIANGLES, 0, 3); // drawing one triangle
  gl.glBindVertexArray(0);
}
```

**Program listing 2.6:** The scene methods – initialisation and rendering

# 3 Element Buffer Objects

When you read the rest of Joey de Vries's example at https://learnopengl.com/#!Getting-started/Hello-Triangle, you will see that he goes on to discuss Element Buffer Objects (EBOs). Program S03.java uses EBOs to render two triangles. The changes to the programs in Section 2 are mainly confined to two methods: fillBuffers() and render(). Program listing 3.1 gives the fillBuffers() method. Program listing 3.2 shows the changes necessary to the render() method to use EBOs. You should be able to work out what is going on in the JOGL code by following Joey's example.

As I have indicated above, a lot of the code for dealing with GPUs can be copied from one program to the next, even with more vertices to deal with. We will see this as we progress through the exercise sheets in later weeks.

## Exercises

1. Instead of filling the triangle that is drawn, a 'wireframe' of it can be drawn by configuring OpenGL to draw lines for the edges of triangles instead of filling the triangles. This is done inside method init(), where other OpenGL attributes are also similarly configured, e.g. setting the clear colour. The C version for achieving this is: `glPolygonMode(GL_FRONT_AND_BACK, GL_LINE)`. The equivalent JOGL version of this is `gl.glPolygonMode(GL.GL_FRONT_AND_BACK, GL3.GL_LINE);` Add this to the method init to see the effect on drawing the triangle. (Note: some OpenGL attributes can be referred to in JOGL using GL.XYZ. This applies to attributes that have been available in all previous versions of OpenGL. However, some require a specific OpenGL profile to be used, e.g. GL3.GL_LINE. In most cases, you can resort to just using GL3.XYZ for all of them, unless you are using a later OpenGL 4.x profile.) The default is glPolygonMode(GL_FRONT_AND_BACK, GL_FILL).
2. Change the program to draw three triangles.
3. (Advanced): Look through the solutions for exercise 2 and 3 at https://learnopengl.com/#!Getting-started/Hello-Triangle.

```
// ************************************************
  /* THE BUFFERS
   */

  private int[] vertexBufferId = new int[1];
  private int[] vertexArrayId = new int[1];
  private int[] elementBufferId = new int[1];

  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 = 3;
    int numVertexFloats = 3;
    int offset = 0
    gl.glVertexAttribPointer(0, numVertexFloats, GL.GL_FLOAT, false,
                             stride*Float.BYTES, offset);
    gl.glEnableVertexAttribArray(0);
    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 3.1:** Setting up the buffers with the addition of EBOs

```
  // ************************************************
  /* THE SCENE
   * Now define all the methods to handle the scene.
   */

  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 3.2:** Using EBOs for rendering