ITECH3229 - Week 9 – Polygons, Colours & Textures in OpenGL ES

Introduction

This lab will guide you through the creation of a number of projects which use the OpenGL ES library in Android to draw a coloured and textured 2D polygons. Use the source code and XML provided (feel free to copy and paste from the electronic version of this document) to create the projects as guided.

There are four guided projects this week, after which there are two projects where you're asked to modify code to produce a desire result.

Project 1 of 4 - OpenGL ES Triangle

Create a new **Android Application** called **Lab 9 – Triangle**, a company URL of **federation.edu.au** and a minimum API of **19** and using an **Empty Activity** as default.

We don't want the title bar or to display in our application, so replace the **@android:style/Theme** line in your manifest with the following line:

```
android:theme="@android:style/Theme.AppCompat.Light.NoActionBar"
```

Now place the following code inside your **MainActivity.java** file:

```
// Your package-name here!

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.opengl.GLSurfaceView;
import android.util.DisplayMetrics;

public class MainActivity extends AppCompatActivity
{
    private GLSurfaceView glSurface;
    private MyRenderer    myRenderer;

    public static float screenWidth, screenHeight;

    @Override
    protected void onCreate(Bundle savedInstanceState)
    {
        // Call the superclass onCreate
        super.onCreate(savedInstanceState);

        // Get the screen size
        DisplayMetrics displaymetrics = new DisplayMetrics();
        getWindowManager().getDefaultDisplay().getMetrics(displaymetrics);
        screenWidth  = displaymetrics.heightPixels;
        screenHeight = displaymetrics.widthPixels;

        // Instantiate our GLSurfaceView passing it this activity as the context
        glSurface = new GLSurfaceView(this);

        // Instantiate our renderer instance so we can use it to draw things
        myRenderer = new MyRenderer();

        // Specify that we should actually use our myRenderer instance to draw things!
        glSurface.setRenderer(myRenderer);
```

```
        // Display the surface!
        setContentView(glSurface);
    }

    @Override
    protected void onPause()
    {
        // When the application is paused, we should call both Activity's
        // and GLSurfaceView's onPause() methods.
        super.onPause();
        glSurface.onPause();
    }

    @Override
    protected void onResume()
    {
        // When the application is resumed after pausing, we should call
        // both Activity's and GLSurfaceViews onResume() methods.
        super.onResume();
        glSurface.onResume();
    }
}
```

Now we'll write the **MyRenderer** class, so create a new class called **MyRenderer** and enter the following code:

```
// Your package name here!

import javax.microedition.khronos.opengles.GL10;
import javax.microedition.khronos.egl.EGLConfig;
import android.opengl.GLSurfaceView.Renderer;

import static javax.microedition.khronos.opengles.GL10.*;

public class MyRenderer implements Renderer
{
   private Triangle t;

   @Override
   public void onSurfaceCreated(GL10 gl, EGLConfig config)
   {
      System.out.println("In onSurfaceCreated!");

      // Set the clear colour to black at full opacity
      gl.glClearColor(0.0f, 0.0f, 0.0f, 1.0f);

      // Set our drawing colour to red at full opacity
      gl.glColor4f(1.0f, 0.0f, 0.0f, 1.0f);
   }

   // Method to reset the surface if it changes
   @Override
   public void onSurfaceChanged(GL10 gl, int width, int height)
   {
      System.out.println("In onSurfaceChanged!");

      // Update our static screen width and height values
      MainActivity.screenWidth  = width;
      MainActivity.screenHeight = height;

      // Instantiate our triangle
      t = new Triangle();

      // Reset the width and height of our viewport
      gl.glViewport(0, 0, width, height);
```

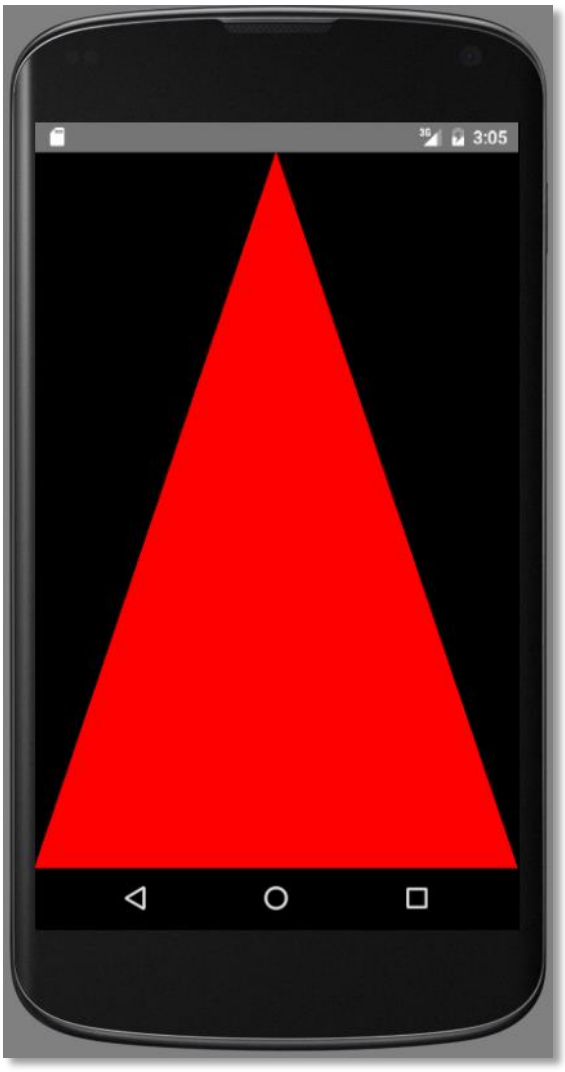
```
        // Reset the Projection matrix and the set it up with an orthographic (2D) projection
        // Parameters: Left, Right, Bottom, Top, Near, Far
        // Note: If we wanted our origin to be in the top-left instead of the bottom-left we can
        // just exchange the bottom and top values here!
        gl.glMatrixMode(GL_PROJECTION);
        gl.glLoadIdentity();
        gl.glOrthof(0, width, 0, height, 1, -1);

        // Switch to the ModelView matrix and reset it
        gl.glMatrixMode(GL_MODELVIEW);
        gl.glLoadIdentity();
    }

    // Method to draw the frame
    @Override
    public void onDrawFrame(GL10 gl)
    {
        // Clear the screen
        gl.glClear(GL10.GL_COLOR_BUFFER_BIT);

        // Enable the vertex client state
        gl.glEnableClientState(GL_VERTEX_ARRAY);

        // Draw the triangle!
        t.draw(gl);

        // Disable the vertex client state
        gl.glDisableClientState(GL_VERTEX_ARRAY);
    }
}
```

Finally, we'll create our **Triangle** class itself which defines a triangle model and a method to draw it:

```
// Your package-name here!

import java.nio.ByteBuffer;
import java.nio.ByteOrder;
import java.nio.FloatBuffer;

import javax.microedition.khronos.opengles.GL10;
import static javax.microedition.khronos.opengles.GL10.*;

public class Triangle
{
    public static final int VERTEX_COUNT      = 3; // A triangle has 3 vertices
    public static final int COORDS_PER_VERTEX = 2; // Only x and y coords for 2D!
    public static final int BYTES_PER_FLOAT = Float.SIZE / 8; // 4 bytes per float

    // Buffer size is 3 * 2 * 4 = 24 Bytes
    public static final int BUFFER_SIZE = VERTEX_COUNT * COORDS_PER_VERTEX * BYTES_PER_FLOAT;

    private float vertices[] = {
                                    0.0f, 0.0f,                         // Bottom-left
                MainActivity.screenWidth , 0.0f,                        // Bottom-right
            MainActivity.screenWidth / 2.0f, MainActivity.screenHeight }; // Top-middle

    private FloatBuffer vertexBuffer;

    // Constructor
    public Triangle()
    {
        // Allocate memory for our ByteBuffer (3 * 2 * 4 = 24 Bytes)
        ByteBuffer byteBuffer = ByteBuffer.allocateDirect(BUFFER_SIZE);

        // Specify byte order in use (little-endian or big-endian)
        byteBuffer.order(ByteOrder.nativeOrder());
```

```
        // Specify our vertexBuffer as a FloatBuffer version
        // of our byteBuffer object
        vertexBuffer = byteBuffer.asFloatBuffer();

        // Put our vertex data in the vertex buffer
        vertexBuffer.put(vertices);

        // Reset the start of the buffer to 0 and mark the end position as the end!
        vertexBuffer.flip();
    }

    void draw(GL10 gl)
    {
        // Specify our vertexes. Params: number of values, type, stride, data source
        gl.glVertexPointer(Triangle.COORDS_PER_VERTEX, GL_FLOAT, 0, vertexBuffer);

        // And finally draw the triangle
        // Params: Primitive type, start vertex number, end vertex number
        gl.glDrawArrays(GL_TRIANGLES, 0, Triangle.VERTEX_COUNT);
    }
}
```

Once all this has been done, run the application in an AVD (Android Virtual Device) or on a physical device. Once running, the application should look like this:



Project 2 of 4 - Multicoloured Triangle

Create a new **Android Application** with called **Lab 9 – Multicoloured Triangle** – we'll use the same MainActivity code as the first project, and our MyRenderer class will only be different in that we create and draw a **ColourTriangle** object, which we'll set up shortly. Here's the renderer code:

```
// Your package-name here!

import android.opengl.GLSurfaceView.Renderer;
import javax.microedition.khronos.egl.EGLConfig;
import javax.microedition.khronos.opengles.GL10;
import static javax.microedition.khronos.opengles.GL10.*;

public class MyRenderer implements Renderer
{
   private ColourTriangle triangle;

   @Override
   public void onSurfaceCreated(GL10 gl, EGLConfig config)
   {
      // Set the clear colour to black at full opacity
      gl.glClearColor(0.0f, 0.0f, 0.0f, 1.0f);

      // We don't specify a single drawing colour anymore - our colour data
      // is in our vertex buffer along with our vertex locations!
   }

   // Method to reset the surface if it changes
   @Override
   public void onSurfaceChanged(GL10 gl, int width, int height)
   {
      // Update our static screen width and height values
      MainActivity.screenWidth  = width;
      MainActivity.screenHeight = height;

      // Instantiate our triangle
      triangle = new ColourTriangle();

      // Reset the width and height of our viewport
      gl.glViewport(0, 0, width, height);

      // Reset the Projection matrix and the set it up with an orthographic (2D) projection
      // Parameters: Left, Right, Bottom, Top, Near, Far
      // Note: If we wanted our origin to be in the top-left instead of the bottom-left we can
      // just exchange the bottom and top values here!
      gl.glMatrixMode(GL_PROJECTION);
      gl.glLoadIdentity();
      gl.glOrthof(0, width, 0, height, 1, -1);

      // Switch to the ModelView matrix and reset it
      gl.glMatrixMode(GL_MODELVIEW);
      gl.glLoadIdentity();
   }

   // Method to draw the frame
   @Override
   public void onDrawFrame(GL10 gl)
   {
      // Clear the screen
      gl.glClear(GL10.GL_COLOR_BUFFER_BIT);

      // Enable the vertex and colour array client states
      gl.glEnableClientState(GL_VERTEX_ARRAY);
      gl.glEnableClientState(GL_COLOR_ARRAY);
```

```
        // Draw the triangle!
        triangle.draw(gl);

        // Disable the vertex and colour array client states
        gl.glDisableClientState(GL_VERTEX_ARRAY);
        gl.glDisableClientState(GL_COLOR_ARRAY);
    }
}
```

Finally, create a new class called **ColourTriangle** and provide it with the following code:

```
package au.edu.federation.lab9b_multicoloured_triangle;


import java.nio.ByteBuffer;
import java.nio.ByteOrder;
import java.nio.FloatBuffer;

import javax.microedition.khronos.opengles.GL10;
import static javax.microedition.khronos.opengles.GL10.*;

public class ColourTriangle
{
    // We're drawing 1 triangle, so there are only 3 vertices in it
    static final int NUM_VERTICES       = 3;

    // For 2D, each vertex will have an x and y component only
    static final int COORDS_PER_VERTEX  = 2;

    // Each vertex will have a Red, Green, Blue and Alpha components
    static final int COLOURS_PER_VERTEX = 4;

    // Get the size of a float in bytes
    static final int BYTES_PER_FLOAT    = Float.SIZE / 8; // 4 Bytes

    // Work out the size of each vertex in bytes (2 + 4) * 4 = 24 Bytes per vertex
    static final int VERTEX_SIZE_BYTES  = (COORDS_PER_VERTEX + COLOURS_PER_VERTEX) *
BYTES_PER_FLOAT;

    // Work out the size of our ByteBuffer in bytes (24 * 3 = 72 Bytes buffer size)
    static final int BYTE_BUFFER_SIZE = VERTEX_SIZE_BYTES * NUM_VERTICES;

    static FloatBuffer vertexBuffer;

    public ColourTriangle()
    {
        // Allocate memory for our byte buffer
        ByteBuffer byteBuffer = ByteBuffer.allocateDirect(BYTE_BUFFER_SIZE);

        // Specify the byte ordering (BIG_ENDIAN or LITTLE_ENDIAN)
        byteBuffer.order(ByteOrder.nativeOrder());

        // Specify our vertices to be the required size in floats
        vertexBuffer = byteBuffer.asFloatBuffer();

        // Our FloatBuffer will have 18 floats (0 through 17), because it's 3
        //vertices, and 6 values per vertex
        vertexBuffer.put( new float[] {
            //                              x,                              y,    r,    g,    b,    a
                                         0.0f,                           0.0f, 1.0f, 0.0f, 0.0f, 1.0f,
                        MainActivity.screenWidth,                           0.0f, 0.0f, 1.0f, 0.0f, 1.0f,
                    MainActivity.screenWidth / 2.0f, MainActivity.screenHeight, 0.0f, 0.0f, 1.0f, 1.0f  }
);

        // Reset the vertices buffer ready for use
        vertexBuffer.flip();
```

```
    }

    public void draw(GL10 gl)
    {
        // Starting from position 0, there are 2 vertex positions (x and y),
        // they're floats, the stride is VERTEX_SIZE_BYTES bytes between verts
        // and we're drawing the data from our vertexBuffer
        vertexBuffer.position(0);
        gl.glVertexPointer(COORDS_PER_VERTEX, GL_FLOAT, VERTEX_SIZE_BYTES, vertexBuffer);

        // Starting from position 2 (the third position), there are 4 vertex
        // positions (r/g/b/a), they're also floats, the stride is still
        // VERTEX_SIZE_BYTES bytes between vertexes, and we're still taking our
        // data from the same vertexBuffer
        vertexBuffer.position(2);
        gl.glColorPointer(COLOURS_PER_VERTEX, GL_FLOAT, VERTEX_SIZE_BYTES, vertexBuffer);

        // Draw our geometry using triangles, starting at vertex 0, and that
        // our triangle model simply consists of  3 vertices
        gl.glDrawArrays(GL_TRIANGLES, 0, NUM_VERTICES);
    }
}
```

With all that done we can finally launch the application on your AVD or Android device - it should end up looking like this:

OpenGL has ***interpolated*** the colours between vertices to give us nice, smooth gradients between the different colour vertices! Pretty colours! =D

Project 3 of 4 - Drawing with Indices

Create a new **Android Application** called **Lab 9 – Multicoloured Rectangle Indices**.The **MainActivity** class is identical to the code we used in our previous projects except for the different package name.

The **MyRenderer** class is identical to the previous MyRenderer, except this time we create and draw a **ColourRectangle** object instead of a **ColourTriangle** object.

As before, set the activity to be without a TitleBar and to display in Fullscreen by adding the following line to the manifest in the application section:

```
android:theme="@android:style/Theme.AppCompat.Light.NoActionBar"
```

Now for our **ColourRectangle** class itself - this class uses a different way of drawing graphics than you have previously seen – this time we're going to use a set of vertices along with a set of **indices** to indicate which vertices make up each polygon (just a single rectangle, in this case). As such we're going to be using the **glDrawElements** method instead of **glDrawArrays** to draw our geometry.

Please look at the code carefully to see how we store both vertices and indices for the rectangle (which we'll make out of two triangles):

```
// Your package-name here!

import java.nio.ByteBuffer;
import java.nio.ByteOrder;
import java.nio.FloatBuffer;
import java.nio.ShortBuffer;

import javax.microedition.khronos.opengles.GL10;
import static javax.microedition.khronos.opengles.GL10.*;

public class ColourRectangle
{
   // We're drawing 2 triangles (which make up a quad - i.e. rectangle), and we can do this with 4
vertices
   static final int NUM_VERTICES       = 4;

   // We're going to draw 2 triangles, where each triangle has 3 vertices
   static final int NUM_INDICES        = 6; // 2 * 3 = 6

   // For 2D, each vertex will have an x and y component only
   static final int COORDS_PER_VERTEX  = 2;

   // Each vertex will have a Red, Green, Blue and Alpha components
   static final int COLOUR_COMPONENTS_PER_VERTEX = 4;

   // Get the size of a Float and Short in bytes
   static final int BYTES_PER_FLOAT    = Float.SIZE / 8; // 4 Bytes
   static final int BYTES_PER_SHORT    = Short.SIZE / 8; // 2 Bytes

   // Work out the size of each vertex in bytes (2 + 4) * 4 = 24 Bytes per vertex
   static final int VERTEX_SIZE_BYTES  = (COORDS_PER_VERTEX + COLOUR_COMPONENTS_PER_VERTEX) *
BYTES_PER_FLOAT;

   // Work out the size of our vertex byte buffer (4 vertices * 24 bytes per vertex = 96 bytes)
   static final int VERTEX_BYTE_BUFFER_SIZE = NUM_VERTICES * VERTEX_SIZE_BYTES;

   // Work out the size of our index buffer (6 indices * 2 bytes per short = 12 bytes)
```

```
    static final int INDEX_BYTE_BUFFER_SIZE  = NUM_INDICES * BYTES_PER_SHORT;

    FloatBuffer vertexBuffer;
    ShortBuffer indexBuffer;

    public ColourRectangle()
    {
        // ----- Vertex Buffer Setup -----

        // Allocate memory for our byte buffer
        ByteBuffer byteBuffer = ByteBuffer.allocateDirect(VERTEX_BYTE_BUFFER_SIZE);

        // Specify the byte ordering (BIG_ENDIAN or LITTLE_ENDIAN)
        byteBuffer.order(ByteOrder.nativeOrder());

        // Set up our vertex buffer as a float version of the byte buffer
        vertexBuffer = byteBuffer.asFloatBuffer();

        // Our FloatBuffer will have 18 floats (0 through 17), because it's 3
        // vertices, and 6 values per vertex
        vertexBuffer.put( new float[] {
            //                              x,                              y,    r,    g,    b,    a
                                          0.0f,                           0.0f, 1.0f, 0.0f, 0.0f, 1.0f,
// Bottom left  - vertex 0, red
                                           MainActivity.screenWidth,      0.0f, 0.0f, 1.0f, 0.0f, 1.0f,
// Bottom right - vertex 1, green
                                                0.0f, MainActivity.screenHeight, 0.0f, 0.0f, 1.0f, 1.0f,
// Top left     - vertex 2, blue
                                         MainActivity.screenWidth, MainActivity.screenHeight, 0.0f, 0.0f, 0.0f, 1.0f  } );
// Top right    - vertex 3, black


        // ----- Index Buffer Setup -----

        // Allocate memory for our byte buffer
        byteBuffer = ByteBuffer.allocateDirect(INDEX_BYTE_BUFFER_SIZE);

        // Specify the byte ordering (BIG_ENDIAN or LITTLE_ENDIAN)
        byteBuffer.order(ByteOrder.nativeOrder());

        // Set up our index buffer as a short version of the byte buffer
        indexBuffer = byteBuffer.asShortBuffer();

        indexBuffer.put( new short[] {
              0, 1, 2,     // The first  triangle is made up of vertices 0, 1 and 2
              1, 2, 3 } ); // The second triangle is made up of vertices 1, 2 and 3

        // Reset the vertices buffer ready for use
        indexBuffer.flip();
    }

    public void draw(GL10 gl)
    {
        // Starting from position 0, there are 2 vertex positions (x and y),
        // they're floats, the stride is VERTEX_SIZE_BYTES bytes between verts
        // and we're drawing the data from our vertexBuffer
        vertexBuffer.position(0);
        gl.glVertexPointer(COORDS_PER_VERTEX, GL_FLOAT, VERTEX_SIZE_BYTES, vertexBuffer);

        // Starting from position 2 (the third position), there are 4 vertex
        // colours (r/g/b/a), they're also floats, the stride is still
        // VERTEX_SIZE_BYTES bytes between vertices, and we're still taking our
        // data from the same vertexBuffer
        vertexBuffer.position(2);
        gl.glColorPointer(COLOUR_COMPONENTS_PER_VERTEX, GL_FLOAT, VERTEX_SIZE_BYTES, vertexBuffer);

        // Draw our geometry from specifying the indices to use
```

```
        gl.glDrawElements(GL_TRIANGLES, NUM_INDICES, GL_UNSIGNED_SHORT, indexBuffer);
    }
}
```

When you then run the program you should see this:

Project 4 of 4 - Textured Quad

In our final guided project we'll create a textured rectangle (i.e. a "quad"-rilateral). So create a project called **Lab 9 - Textured Rectangle** and use the following code for your MainActivity. It's very similar to the previous MainActivity code but this time we have a **getContext()** method which will return us the current application context which we need to load the texture!

```
// Your package-name here!

import android.opengl.GLSurfaceView;
import android.os.Bundle;
import android.app.Activity;
import android.content.Context;

public class MainActivity extends Activity
{
        public static Context context;
        private GLSurfaceView glSurface;
        private MyRenderer    myRenderer;

        @Override
```

```
        protected void onCreate(Bundle savedInstanceState)
        {
                // Call the superclass onCreate
                super.onCreate(savedInstanceState);


                // Get a copy of the application context
                context = getApplicationContext();

                // Instantiate our GLSurfaceView passing it this activity as the context
                glSurface = new GLSurfaceView(this);

                // Instantiate our renderer instance so we can use it to draw things
                myRenderer = new MyRenderer();

                // Specify that we should actually use our myRenderer instance to draw things!
                glSurface.setRenderer(myRenderer);

                // Display the surface!
                setContentView(glSurface);
        }

        @Override
        protected void onPause()
        {
                // When the application is paused, we should call both Activity's
                // and GLSurfaceView's onPause() methods.
                super.onPause();
                glSurface.onPause();
        }

        @Override
        protected void onResume()
        {
                // When the application is resumed after pausing, we should call
                // both Activity's and GLSurfaceViews onResume() methods.
                super.onResume();
                glSurface.onResume();
        }


        // Method to return the application context
        public static Context getContext()
        {
              return context;
        }
}
```

By default we don't have an **Assets** folder in the top level of our project, you can either manually create one, or right click on your project and choose: **New | Folder | Assets Folder**. Once you've create the assets folder, copy the provided **bricks-256x256.jpg** and **dog.jpg** images into the folder so we can load them as textures.

Okay – let's do the renderer, which uses the **GL_TEXTURE_COORD_ARRAY** vertex pointer and ***specifically enables 2D textures***:

```
// Your package-name here!

import javax.microedition.khronos.opengles.GL10;
import javax.microedition.khronos.egl.EGLConfig;
```

```
import android.opengl.GLSurfaceView.Renderer;
import static javax.microedition.khronos.opengles.GL10.*;

public class MyRenderer implements Renderer
{
   private TextureRectangle t;

   @Override
   public void onSurfaceCreated(GL10 gl, EGLConfig config)
   {
      t = new TextureRectangle(gl);
   }

   // Method to reset the surface if it changes (i.e. flips vert to horizontal etc).
   @Override
   public void onSurfaceChanged(GL10 gl, int width, int height)
   {
      // Update our static screen width and height values
      MainActivity.screenWidth  = width;
      MainActivity.screenHeight = height;

      // Reset the width and height of our viewport
      gl.glViewport(0, 0, width, height);

      // Reset the Projection matrix and the set it up with an orthographic (2D) projection
      // Parameters: Left, Right, Bottom, Top, Near, Far
      // Note: If we wanted our origin to be in the top-left instead of the bottom-left we can
      // just exchange the bottom and top values here!
      gl.glMatrixMode(GL_PROJECTION);
      gl.glLoadIdentity();
      gl.glOrthof(0, width, 0, height, 1, -1);

      // Switch to the ModelView matrix and reset it
      gl.glMatrixMode(GL_MODELVIEW);
      gl.glLoadIdentity();

      // Enable 2D texturing
      gl.glEnable(GL10.GL_TEXTURE_2D);
   }

   // Method to draw the frame
   @Override
   public void onDrawFrame(GL10 gl)
   {
      // Set the clear colour and clear the screen
      gl.glClearColor(0.0f,  0.0f,  0.0f,  1.0f);
      gl.glClear(GL10.GL_COLOR_BUFFER_BIT);

      // Enable the client states
      gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
      gl.glEnableClientState(GL10.GL_TEXTURE_COORD_ARRAY);

      // Rotate on the X axis.
      // Params: Rotation amount in degrees, x axis direction, y axis direction, z axis direction
      //gl.glRotatef(0.7f, 0.0f, 0.0f, 1.0f);

      // Draw the textured rectangle
      t.draw(gl);

      // Disable the client state before leaving
      gl.glDisableClientState(GL10.GL_VERTEX_ARRAY);
      gl.glDisableClientState(GL10.GL_TEXTURE_COORD_ARRAY);
   }

}
```

Now that the MainActivity and the renderer are complete, delete the **ColourRectangle** class and replace it with a **TextureRectangle** class with the following contents:

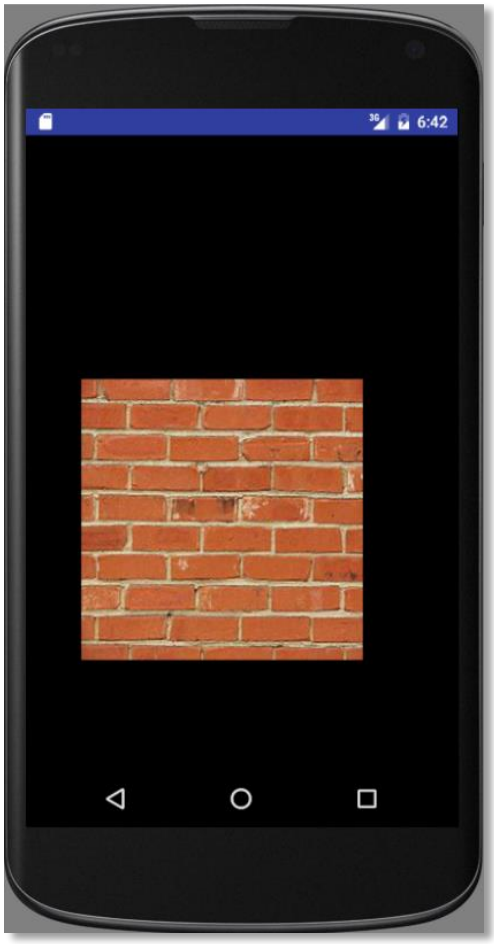
```
// Your package-name here!

import java.io.IOException;
import java.io.InputStream;
import java.nio.ByteBuffer;
import java.nio.ByteOrder;
import java.nio.FloatBuffer;
import java.nio.ShortBuffer;

import javax.microedition.khronos.opengles.GL10;
import static javax.microedition.khronos.opengles.GL10.*;

import android.content.Context;
import android.content.res.AssetManager;
import android.graphics.Bitmap;
import android.graphics.BitmapFactory;
import android.graphics.Matrix;
import android.opengl.GLUtils;

public class TextureRectangle
{
   // We'll keep a copy of the context around so we can use it to access the AssetManager
   static Context context;

   // We'll also keep a copy of the GL10 object around
   static GL10 gl;

   // We're drawing a rectangle comprised of 2 triangles, which have 4 vertices
   // combined (because we're re-using vertices!)
   static final int NUM_VERTICES = 4;

   // We'll draw two triangles as a TRIANGLE_STRIP, where the first three
   // vertices form the first triangle, then the next vertex along with the
   // previous two vertices form the second triangle. So our index data
   // will be 0 -> 1 -> 2 -> 3, which is 4 values.
   static final int NUM_INDICES = 4;

   // For 2D, each vertex will have an x and y component only
   static final int COORDS_PER_VERTEX = 2;

   // Each vertex will have s and t components for a 2D texture
   static final int TEX_COORDS_PER_VERTEX = 2;

   // Get the size of a Float and Short in bytes
   static final int BYTES_PER_FLOAT = Float.SIZE / 8; // 4 bytes per float
   static final int BYTES_PER_SHORT = Short.SIZE / 8; // 2 bytes per short

   // Work out the size of each vertex in bytes (2 + 2) * 4 = 16 Bytes per
   // vertex
   static final int VERTEX_SIZE_BYTES = (COORDS_PER_VERTEX + TEX_COORDS_PER_VERTEX) *
BYTES_PER_FLOAT;

   // Work out the size of our ByteBuffer in bytes
   static final int VERTEX_BUF_SIZE = NUM_VERTICES * VERTEX_SIZE_BYTES; // 4 * 16 = 64 Bytes
   static final int INDEX_BUF_SIZE = NUM_INDICES * BYTES_PER_SHORT;     // 4 * 2 = 8 Bytes

   // Buffers for our vertices and indices
   static FloatBuffer vertexBuffer;
   static ShortBuffer indexBuffer;

   // The Id value (i.e. handle) for the texture
   int textureId;

   public TextureRectangle(GL10 glInstance)
```

```
    {
        // Get a copy of the MainActivity context
        context = MainActivity.getContext();

        // Take a copy of our GL10 object
        gl = glInstance;

        // Allocate memory for our byte buffer
        ByteBuffer vertexByteBuffer = ByteBuffer.allocateDirect(VERTEX_BUF_SIZE);

        // Specify the byte ordering (BIG_ENDIAN or LITTLE_ENDIAN)
        vertexByteBuffer.order(ByteOrder.nativeOrder());

        // Specify our vertices to be the required size in floats
        vertexBuffer = vertexByteBuffer.asFloatBuffer();

        // Our FloatBuffer will have 16 floats (0 through 15), because it's 4 vertices, and 4 values
per vertex
        //                                  x      y       s     t
        vertexBuffer.put(new float[] { 100.0f, 200.0f, 0.0f, 0.0f,     // Bottom left
                                       600.0f, 200.0f, 1.0f, 0.0f,     // Bottom right
                                       100.0f, 700.0f, 0.0f, 1.0f,     // Top left
                                       600.0f, 700.0f, 1.0f, 1.0f }); // Top right

        // The setup of our vertices is now like this:
        //
        //     2 --- 3
        //     | \   |
        //     |   \ |
        //     0 --- 1

         // Reset the vertex buffer for use
        vertexBuffer.flip();

        // Create a bytebufffer of the desired size for the indices
        ByteBuffer indexByteBuffer = ByteBuffer.allocateDirect(INDEX_BUF_SIZE);

        // Specify the byte order
        indexByteBuffer.order(ByteOrder.nativeOrder());

        // Make the indexBuffer to be a ShortBuffer version of our indexByteBuffer
        indexBuffer = indexByteBuffer.asShortBuffer();

        // Bottom-left -> bottom-right -> top-left -> top-right
        indexBuffer.put(new short[] { 0, 1, 2, 3 } );

        // Reset the index buffer for use
        indexBuffer.flip();

        // Load our texture for use
        textureId = loadGLTexture("bricks-256x256.jpg");
    }

    // Method to load an image, conver it into a texture and return the texture Id
    int loadGLTexture(String filename)
    {
        // Get an AssetManager instance so we can access files in the assets
        // folder
        AssetManager assetManager = MainActivity.getContext().getAssets();

        try
        {
            // Attempt to open the image file
            InputStream inputStream = assetManager.open(filename);

            // Decode the image as a Bitmap object
            Bitmap srcBmp = BitmapFactory.decodeStream(inputStream);
```

```
            // Create a matrix which will flip the bitmap upside down
            Matrix matrix = new Matrix();
            matrix.preScale(1.0f,  -1.0f);

            // Create a new "upside-down corrected" bitmap
            Bitmap newBmp = Bitmap.createBitmap(srcBmp, 0, 0, srcBmp.getWidth(), srcBmp.getHeight(),
matrix, true);

            // Get rid of the memory allocated for the original bitmap
            srcBmp.recycle();

            // Close the image file now we're done with it
            inputStream.close();

            // Generate an array of just a single int for the texture, generate an Id for
            // the texture, and then bind to the texture (so that OpenGL knows that its
            // active and that we're working with it)
            int textures[] = new int[1];
            gl.glGenTextures(1, textures, 0);
            gl.glBindTexture(GL_TEXTURE_2D, textures[0]);

            // Set nearest filtering for minification and magnification
            gl.glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
            gl.glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);

            // Clamp texture to edges (i.e. texture coordinates outside 0.0 to 1.0
            // get clamped to those values)
            gl.glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
            gl.glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);

            // Use the Android GLUtils to build our texture from the decoded Bitmap.
            // Note: To use this we need to import android.opengl.GLUtils;
            GLUtils.texImage2D(GL_TEXTURE_2D, 0, newBmp, 0);

            // Free the bitmap data as we now that have it as a texture!
            newBmp.recycle();

            // Return the Id (i.e. handle) of our texture
            return textures[0];
        }
        catch (IOException ioe)
        {
            ioe.printStackTrace();
        }

        // Return -1 if we failed to load the texture
        return -1;

    } // End of loadGLTexture method

    public void draw(GL10 gl)
    {
        // Starting from position 0, there are 2 vertex positions (x and y),
        // they're floats, and the stride is VERTEX_SIZE_BYTES bytes between
        // vertexes and we're taking our data from the vertexBuffer object
        vertexBuffer.position(0);
        gl.glVertexPointer(COORDS_PER_VERTEX, GL_FLOAT, VERTEX_SIZE_BYTES, vertexBuffer);

        // Starting from position 2 (the third position), there are 2 vertex positions (s/t),
        // they're also floats, the stride is still/ VERTEX_SIZE_BYTES bytes between vertexes
        // and we're still taking our data from the/ vertexBuffer
        vertexBuffer.position(2);
        gl.glTexCoordPointer(TEX_COORDS_PER_VERTEX, GL_FLOAT, VERTEX_SIZE_BYTES, vertexBuffer);

        // Bind to our texture
        gl.glBindTexture(GL_TEXTURE_2D, textureId);
```

```
        // Draw our geometry using the indices specified
        gl.glDrawElements(GL_TRIANGLE_STRIP, NUM_INDICES, GL_UNSIGNED_SHORT, indexBuffer);

    } // End of draw method

} // End of TextureRectangle class
```

Once you place the file **bricks-256x256.jpg** in the **assets** folder and run the application it will look like this:



Try adding a **glRotatef** call in the **draw()** method to rotate the quad at various speeds, like this:

```
// Rotate on the X axis.
// Params: Rotation amount in degrees, x axis direction, y axis direction, z axis direction
gl.glRotatef(0.5f, 0.0f, 0.0f, 1.0f);
```

Try changing the rotation angle to a negative value, what happens?

Also - why does the texture rotate around the bottom left (i.e. the origin)? What could we do to get the rectangle rotating in place? (Think about where the rectangle is in relation to the origin and how we could change that [there's two different ways])

Try changing the texture to a different image (grab one from the web or something!) I've put a **dog.jpg** file along with this lab if you just want to try it quickly… However, the dimensions of the dog.jpg file are not powers-of-2, so it will not work properly and display on devices which do not support non-power-of-2 sized textures!

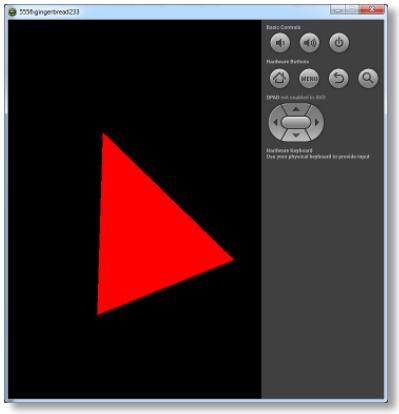
Student Project 1 of 2 - Spinning Triangle

Take your first project from this lab, the red triangle, and modify it so that it rotates in an anti-clockwise direction around the Z axis by 1 degree per frame using the **glRotatef()** method.

You'll notice that the triangle rotates around the origin - which means that most of the time it's off the screen! Modify your code so that the triangle rotates around its centre (as opposed to around the origin). To do this you will have to redefine the vertex locations of the triangle so that it is **centred on the origin**, and then call **glTranslatef** to translate the triangle out into the centre of the screen.

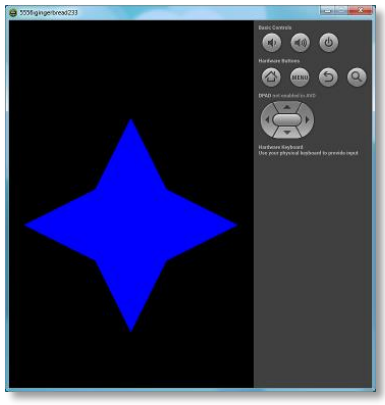
When redefining your triangle vertexes, make the triangle have a width of 240 pixels and a height of 320 pixels.

Once, done your application should look something like this:



Student Project 2 of 2 - Creating a Star

Try to create and display the following "Star" model in OpenGL:



The star has 8 vertices in total, and uses indices to draw the 8 triangles. I chose to take out the colour values from the vertex buffer to just concentrate on the vertex positions and keep things simple – you may choose to do this too. Remember that when we don't have colour data mixed into our vertexBuffer, we just specify a colour to draw in, and then draw something.

You might like to map out your indices from the 8 vertices as specified below (remember to wind your vertices **anti-clockwise** so they face forward!), alternatively, you could not use indices and just duplicate vertex values – as long as you're aware that this isn't terribly efficient!