ITECH3229 - Week 8 – Introduction to 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 variety of animated 2D graphics. 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 is single projects where you're asked to modify code to produce a desire effect.

Don't Forget: Android Studio can automatically format your code, but only if it does not contain errors! Once your code compiles, you can quickly format it by hitting **Ctrl+Alt+L** or from the menu by choosing **Code | Reformat Code…**

Project 1 of 4 – OpenGL ES Setup

The first project we'll look at will demonstrate how to setup OpenGL ES by creating a **GLSurfaceView** object, and then providing a custom **Renderer** class which implements the **onSurfaceCreated**, **onSurfaceChanged** and **onDrawFrame** methods for the GLSurfaceView.

Start by creating a new Android project called **Lab 8 - OpenGL ES Setup** with a company URL (i.e. package name) of **federation.edu.au**. Use API **25** as the **target API** and **19** as the **minimum API** and select an **Empty Activity** to start with.

We'll start by writing our custom **MyRenderer** class (which implements methods of the Renderer class) and so provides the **onSurfaceCreated**, **onSurfaceChanged** and **onDrawFrame** methods:

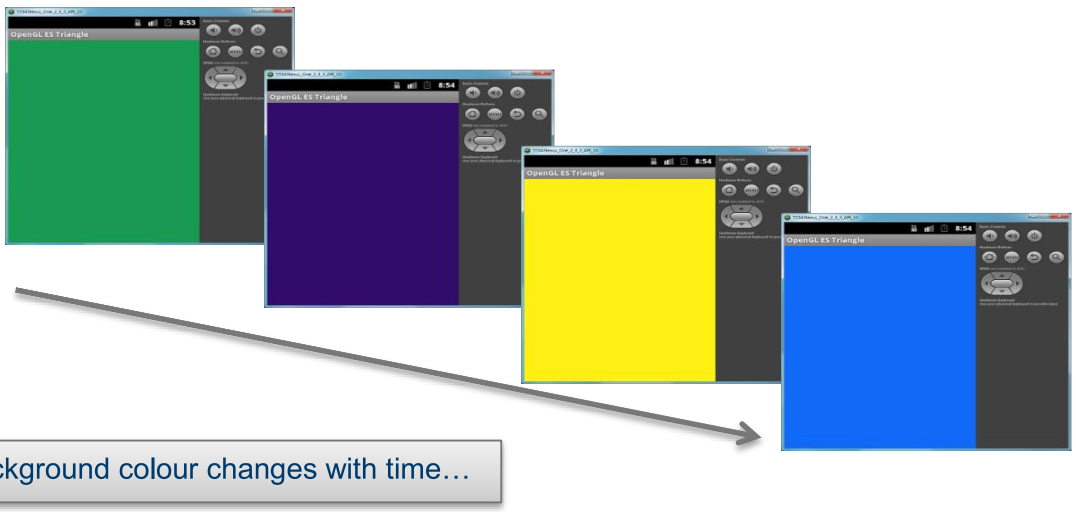
```
// Your package-name here!

import android.opengl.GLSurfaceView.Renderer;

import java.util.Random;

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

// Import OpenGL ES static elements so we don't have to prefix things with GL10.<whatever>
import static javax.microedition.khronos.opengles.GL10.*;

public class MyRenderer implements Renderer
{
   private Random random;

   @Override
   public void onSurfaceCreated(GL10 gl, EGLConfig config)
   {
      // Instantiate our random number generator object
      random = new Random();
   }

   // Method to reset the surface if it changes (i.e. flips vertical to horizontal etc).
   @Override
   public void onSurfaceChanged(GL10 gl, int width, int height)
   {
      // Reset the width and height of our viewport
```

```
        gl.glViewport(0, 0, width, height);

        // Reset the Projection matrix
        gl.glMatrixMode(GL_PROJECTION);
        gl.glLoadIdentity();

        // Set up an orthographic 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
        // or vice-versa we could just exchange the bottom and top values here!
        gl.glOrthof(0, width, 0, height, 1, -1);

        // Reset the Modelview matrix
        gl.glMatrixMode(GL_MODELVIEW);
        gl.glLoadIdentity();
    }

    // Method to draw the frame
    @Override
    public void onDrawFrame(GL10 gl)
    {
        // Specify a random clear colour with full opacity
        // Parameters: Red, Green, Blue, Alpha
        gl.glClearColor(random.nextFloat(), random.nextFloat(), random.nextFloat(), 1.0f);

        // Clear the screen with the colour we specified earlier
        gl.glClear(GL_COLOR_BUFFER_BIT);

        // Sleep for 250ms so we only change colours 4 times per second
        try
        {
              Thread.sleep(250);
          }
        catch (InterruptedException e)
        {
              // Interrupted? Don't worry about it & carry on!
          }

    } // End of onDrawFrame method

} // End of MyRenderer class
```

Now, we can write our **MainActivity** in which we'll create a GLSurfaceView, tell it to use an instance of our MyRenderer class for rendering operations, and display the GLSurfaceView (also, we override the onPause and onResume methods to pause and resume the GLSurfaceView's rendering thread!):

```
// Your package-name here!

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

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

    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 context
        glSurface = new GLSurfaceView(this);

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

        // Specify we'll 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 in that order!
        super.onPause(); // Always call the superclass method first!
        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();// Always call the superclass method first!
        glSurface.onResume();
    }

}
```

Once all this has been done, run the application! Once running, the application should look something like this (colours will be random!):



Project 2 of 4 – Bouncing Points

In our next application, we'll create a Particle class which we'll use to draw some points which move around the screen and bounce off the edges. To begin, create a new Android application called **Lab 8 - Bouncing Points** with the standard settings.

Besides the package name being different, the **MainActivity** class is *identical* to that from the first **OpenGL ES Setup** project, so copy that code across (but not the package name!) from the first project.

Next, we'll create a **Particle** class which stores the x and y (horizontal and vertical) locations of a Particle (i.e. a point) as well as some xSpeed and ySpeed properties so we can move the particles around the screen, and a FloatBuffer in which to hold our vertex data:

```java
// Your package-name here!

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

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

public class Particle
{
   public static final int VERTEX_COUNT      = 1; // Each point is 1 vertex
   public static final int COORDS_PER_VERTEX = 2; // X and Y coords only for 2D!

   // Float.SIZE gives us the size of a float in bits (32 bits), divide by 8
   // to get size in bytes (4 bytes)!
   public static final int BYTES_PER_FLOAT   = Float.SIZE / 8;

    // The size of our buffer (in Bytes) is 1 * 2 * 4 = 8 Bytes
   public static final int BUFFER_SIZE = VERTEX_COUNT * COORDS_PER_VERTEX * BYTES_PER_FLOAT;

    // We'll draw a single vertex at the origin i.e. (0, 0)
   private static float vertices[] = { 0.0f, 0.0f };

   // Our vertexBuffer will hold the vertices data in a format OpenGL can use
   private static FloatBuffer vertexBuffer;

   // Horizontal and vertical location properties
   private float x;
   private float y;

   // Horizontal and vertical speed properties
   private float xSpeed;
   private float ySpeed;

   // Instantiate a Random object so we can place the particles at various random
   // locations and specify that they should move at various random speeds
   private static Random random = new Random();

   // Particle constructor
   public Particle()
   {
      // Allocate memory for our ByteBuffer (1 * 2 * 4 = 8 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 our buffer ready for use
        vertexBuffer.flip();

        // Specify a random position for our point
        x = random.nextFloat() * MainActivity.screenWidth;
        y = random.nextFloat() * MainActivity.screenHeight;

        // Specify random horizontal and vertical speeds for our point
        xSpeed = random.nextFloat() * 5.0f;
        ySpeed = random.nextFloat() * 5.0f;
    }

    void draw(GL10 gl)
    {
        // Reset the ModelView matrix (i.e. the coordinate system for our Particle)
        gl.glLoadIdentity();

        // Specify our vertex pointer to use 2 values (x and y), use floating-point data,
        // use a stride of 0 (i.e. data is tightly packed), and use the vertexBuffer
        // FloatBuffer as the source of data to draw!
        gl.glVertexPointer(2, GL_FLOAT, 0, Particle.vertexBuffer);

        // Translate the coordinate system (i.e. move horizontally and vertically) to the
        // position of the particle, so that even through each point is drawn at (0, 0) -
        // because the coordinate system has moved (0, 0) does not have to be the same
        // as (0, 0) in the screen's coordinate system!
        gl.glTranslatef(x, y, 0.0f);

        // And finally draw the particle as a point
        // Params: Primitive type, start vertex (i.e. 0), end vertex (i.e. 1 -
        // as we're only drawing 1 vertex per particle! Or to think about it another
        // way, the model which we're drawing consists of just 1 vertex!)
        gl.glDrawArrays(GL_POINTS, 0, Particle.VERTEX_COUNT);
    }

    // Method to apply the x and y speeds to the x and y location properties so that
    // the particles move around the screen
    public void update()
    {
        // Move the particle by its component speeds
        x += xSpeed;
        y += ySpeed;

        // Flip the horizontal speed if outside horizontal screen bounds
        if ( (x < 0.0f) || ( x > MainActivity.screenWidth) )
        {
            xSpeed *= -1.0f;
        }

        // Flip the vertical speed if outside vertical screen bounds
        if ( (y < 0.0f) || ( y > MainActivity.screenHeight) )
        {
            ySpeed *= -1.0f;
        }

    } // End of update method

} // End of Particle class
```

Almost done! The final thing to do in this project is to write our **MyRenderer** class which we'll use to instantiate an array of Particle objects, and then in the **onDrawFrame** method we'll tell Android to loop over all the Particles in our array calling the **draw()** and **update**() methods on each one so that they move around the screen:

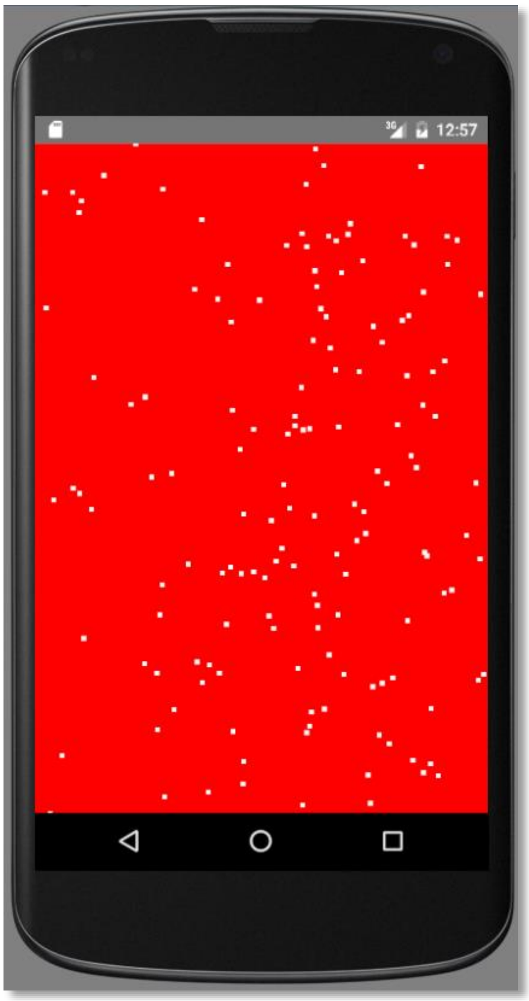
```
// Your package-name here

import android.opengl.GLSurfaceView.Renderer;

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

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

public class MyRenderer implements Renderer
{
   // Define our array of Particle objects
   private Particle particleArray[];

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

      // Specify a "chunky" point size (default is 1.0f)
      gl.glPointSize(8.0f);
   }

   // Method to reset the surface if it flips vertical 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);

      // Instantiate all particles (which use the current screen size values for behaviour)
      particleArray = new Particle[150];
      for (int loop = 0; loop < particleArray.length; loop++)
      {
         particleArray[loop] = new Particle();
      }

      // 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(GL_COLOR_BUFFER_BIT);
```

```
        // Enable the vertex array client state (i.e. tell OpenGL ES that we're using vertices!)
        gl.glEnableClientState(GL_VERTEX_ARRAY);

        // Draw then update all particles in our array
        for (Particle p : particleArray)
        {
           p.draw(gl);
           p.update();
        }

        // Disable the vertex array client state before leaving
        gl.glDisableClientState(GL_VERTEX_ARRAY);
    }

} // End of MyRenderer class
```

Oh, and a final tweak – rather than displaying the titlebar at the top of our Activity – let's go full screen – to do this, open the file **/manifests/AndroidManifest.xml** and change the theme to the following setting:

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

With all that done, when you run the application we should have a number of Particles happily bouncing around the screen in a full-screen activity, as shown below:



Project 3 of 4 – Particle Fountain

Our next application is simply a modification of project 2 so that it displays a particle fountain effect. You can create a new project for this, or simply modify your existing project 2 - thethe only class that changes will be the **Particle** class.

We're going to modify the particle class so that five things are different:

1. Rather than starting at a random location, particles now start off at the centre of the screen.
2. Instead of having a completely random velocity (i.e. xSpeed and ySpeed), particles will have a random x speed so that they may be moving left or right, but will always have a positive y speed so they always start going ***upwards***.
3. The **vertical speed** of the particles will have a **gravity** value applied to them so that they accelerate downwards (this means they'll start to move upwards slower and slower, until they stop moving upwards and start moving downwards faster and faster).
4. The **horizontal speed** of the particles will be damped (i.e. decreased) slightly each frame to simulate air-resistance.
5. Finally, when a particle goes off the bottom of the screen (y > 0) it will be reset to the centre of the screen and the xSpeed and ySpeed values will be re-randomised to new, random values as per step **2**.

To do this, replace the code in the Particle class of your project with the following code – be sure to read through the code rather than just blindly copying and pasting so that you understand how it works!

```
// Your package-name here!

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

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

public class Particle
{
   public static final int VERTEX_COUNT      = 1; // Each point is 1 vertex
   public static final int COORDS_PER_VERTEX = 2; // X and Y coords only for 2D!

   // Float.SIZE gives us the size of a float in bits (32 bits), divide by 8
   // to get size in bytes (4 bytes)!
   public static final int BYTES_PER_FLOAT   = Float.SIZE / 8;

    // The size of our buffer (in Bytes) is 1 * 2 * 4 = 8 Bytes
   public static final int BUFFER_SIZE = VERTEX_COUNT * COORDS_PER_VERTEX * BYTES_PER_FLOAT;

    // We'll draw a single vertex at the origin i.e. (0, 0)
   private static float vertices[] = { 0.0f, 0.0f };

   // Our vertexBuffer will hold the vertices data in a format OpenGL can use
   private static FloatBuffer vertexBuffer;

   // Horizontal and vertical location properties
   private float x;
   private float y;

   // Horizontal and vertical speed properties
```

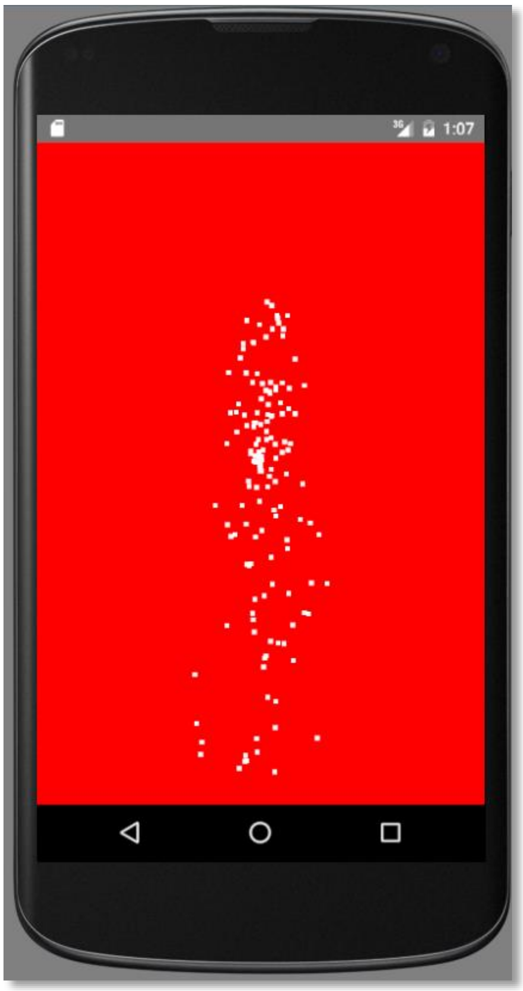
```
    private float xSpeed;
    private float ySpeed;

    // Instantiate a Random object so we can place the particles at various random
    // locations and specify that they should move at various random speeds
    private static Random random = new Random();

    private static float GRAVITY = 0.04f;

    // Particle constructor
    public Particle()
    {
       // Allocate memory for our ByteBuffer (1 * 2 * 4 = 8 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 our buffer ready for use
       vertexBuffer.flip();

       // Set our point location to be the centre of the screen
       x = MainActivity.screenWidth  / 2.0f;
       y = MainActivity.screenHeight / 2.0f;

       // Specify random horizontal and vertical speeds for our point
       xSpeed = random.nextFloat() - 0.5f; // Range: -0.5 to + 0.5
       ySpeed = random.nextFloat() * 5.0f; // Always going up
    }

    void draw(GL10 gl)
    {
       // Reset the ModelView matrix (i.e. the coordinate system for our Particle)
       gl.glLoadIdentity();

       // Specify our vertex pointer to use 2 values (x and y), use floating-point data,
       // use a stride of 0 (i.e. data is tightly packed), and use the vertexBuffer
       // FloatBuffer as the source of data to draw!
       gl.glVertexPointer(2, GL_FLOAT, 0, Particle.vertexBuffer);

       // Translate the coordinate system (i.e. move horizontally and vertically) to the
       // position of the particle, so that even through each point is drawn at (0, 0) -
       // because the coordinate system has moved (0, 0) does not have to be the same
       // as (0, 0) in the screen's coordinate system!
       gl.glTranslatef(x, y, 0.0f);

       // And finally draw the particle as a point
       // Params: Primitive type, start vertex (i.e. 0), end vertex (i.e. 1 -
       // as we're only drawing 1 vertex per particle! Or to think about it another
       // way, the model which we're drawing consists of just 1 vertex!)
       gl.glDrawArrays(GL_POINTS, 0, Particle.VERTEX_COUNT);
    }

    // Method to apply the x and y speeds to the x and y location properties so that
    // the particles move around the screen
    public void update()
    {
       // Apply gravity
       ySpeed -= GRAVITY;

       // Move the particle by its component speeds
```

```
        x += xSpeed;
        y += ySpeed;

        // Reset particles if they go off the bottom of the screen
        if (y < 0.0f)
        {
           // Reset our point location to the centre of the screen
           x = MainActivity.screenWidth  / 2.0f;
           y = MainActivity.screenHeight / 2.0f;

           // Specify random horizontal and vertical speeds for our point
           xSpeed = random.nextFloat() - 0.5f; // Range: -0.5 to +0.5
           ySpeed = random.nextFloat() * 5.0f; // Range: 0.0f to 5.0f      }

    } // End of update method

} // End of Particle class
```

With all that done you should have a app that looks something like this (it's hard to really see in as a "fountain" without seeing it move, unfortunately!)

Now that you have it all up and running:

- Try changing the value of **GRAVITY** to lower and higher values!
- Try changing the initial random speeds so the particles can move faster, or slower…
- Try changing the re-spawn location to be somewhere else on the screen!
- Try using **more particles** – how many more? You choose! =D



Project 4 of 4 – Bouncing Lines

Our last guided project will demonstrate how to draw lines which bounce around the screen – in this project we're going to **modify the vertex data itself** – rather than merely moving the coordinate system around to draw things at different locations. Sometimes this is a terrible idea (like if a model had a large number of vertices), but other times we WANT to modify the vertex data… so let's try it out…

Create a new project called **Lab 8 – Bouncing Lines** and copy in the MainActivity code from any previous project we've looked at in this lab. Now, we want to create a **Line** class to handle our line drawing - use the following code for it:

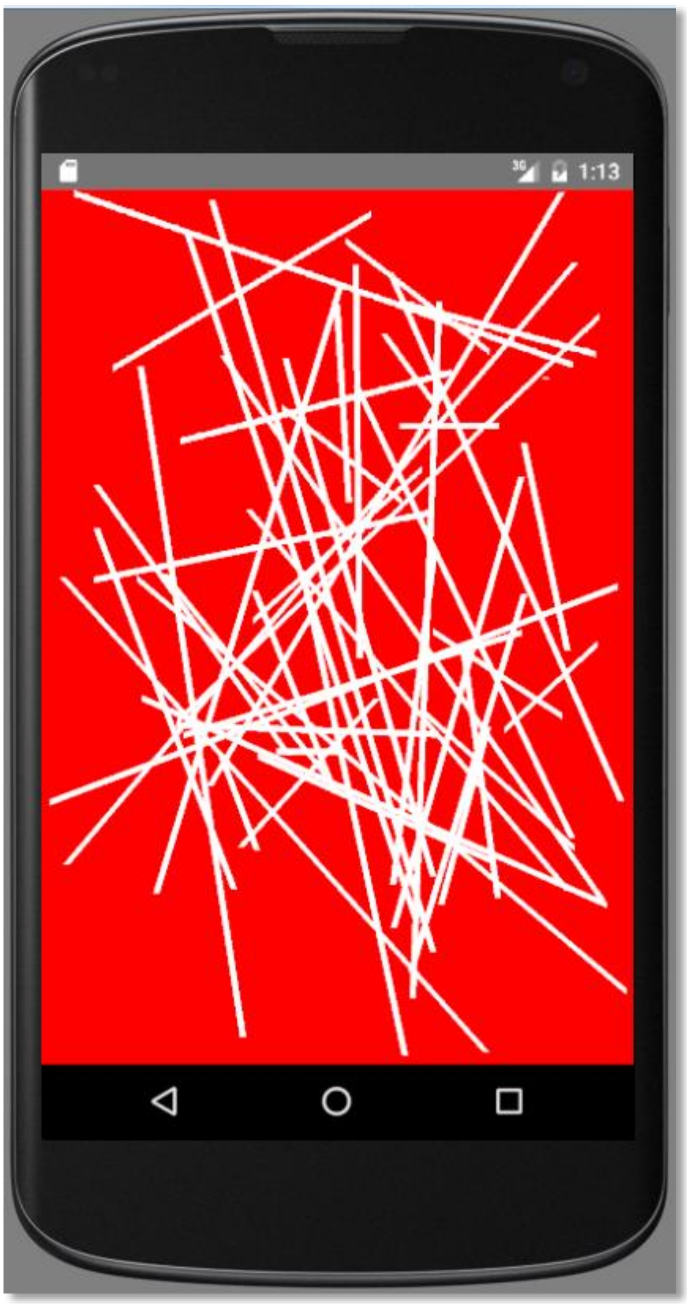
```
// Your package name here

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.*;

import java.util.Random;

public class Line
{
   public static final int VERTEX_COUNT      = 2;               // Each line has 2 vertices
   public static final int COORDS_PER_VERTEX = 2;               // Each vertex has X and Y locations
   public static final int BYTES_PER_FLOAT   = Float.SIZE / 8; // Float.SIZE gives us the size of a
float in bits (32 bits), divide by 8 to get size in bytes (4 bytes)!

    // 2 vertices * 2 values per vertex  * 4 bytes per float  = 16 bytes buffer size
   public static final int BUFFER_SIZE = VERTEX_COUNT * COORDS_PER_VERTEX * BYTES_PER_FLOAT;

    // 2  values per vertex, and 2 vertices means we have 4 floats per Line
   private static float vertices[] =  new float[VERTEX_COUNT * COORDS_PER_VERTEX];

   private static FloatBuffer vertexBuffer;

   private static Random random = new Random();

   // ----- Per-Line Properties -----
   private float x[]      = new float[VERTEX_COUNT];
   private float y[]      = new float[VERTEX_COUNT];
   private float xSpeed[] = new float[VERTEX_COUNT];
   private float ySpeed[] = new float[VERTEX_COUNT];

   // Particle constructor
   public Line()
   {
      // Set all our x and y values to be randomly placed somewhere on the screen
      for (int loop = 0; loop < VERTEX_COUNT; loop ++)
      {
         x[loop]      = random.nextFloat() * MainActivity.screenWidth;
         y[loop]      = random.nextFloat() * MainActivity.screenHeight;

         // Set all out x and y speeds to be random
         xSpeed[loop] = random.nextFloat() * 5.0f;
         ySpeed[loop] = random.nextFloat() * 5.0f;
      }

      // Allocate memory for our ByteBuffer (2 * 2 * 4 = 16 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)
    {
        // No need to reset the ModelView matrix – we haven't modified it!

        // Set the drawing colour to fully opaque white
        gl.glColor4f(1.0f, 1.0f, 1.0f, 1.0f);

        // Specify our vertexes
        gl.glVertexPointer(2, GL_FLOAT, 0, Line.vertexBuffer);

        // And finally draw the line
        // Params: Primitive type, start location of vertexBuffer, end location of vertexBuffer
        gl.glDrawArrays(GL_LINES, 0, Line.VERTEX_COUNT);
    }

    public void update()
    {
        // Update all the vertex locations and reverse speeds if required
        for (int loop = 0; loop < VERTEX_COUNT; loop++)    {

            // Move the endpoints of the line
            x[loop] += xSpeed[loop];
            y[loop] += ySpeed[loop];

            // Flip the horizontal speed if outside horiz screen bounds
            if ( (x[loop] < 0.0f) || ( x[loop] > MainActivity.screenWidth) ) {
                xSpeed[loop] *= -1.0f;
            }

            // Flip the vertical speed if outside vertical screen bounds
            if ( (y[loop] < 0.0f) || ( y[loop] > MainActivity.screenHeight) ) {
                ySpeed[loop] *= -1.0f;
            }
        }

        updateVertices();
    }

    // Method to take our separate x and y values and merge them into the vertices array
    void updateVertices()
    {
        // Merge our separate x and y values into the vertices array
        for (int loop = 0; loop < VERTEX_COUNT; loop++)
        {
            vertices[loop * 2]     = x[loop];
            vertices[loop * 2 + 1] = y[loop];
        }

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

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

}
```

Now replace the contents of the **MyRenderer** class with the following code which will create a number of Line objects and draw & update them each frame:

```
// 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
{
    // Define our array of Particle objects
    private Line lineArray[];
    private static int NUM_LINES = 50;

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

        // Specify a "chunky" line width (default is 1.0f)
        gl.glLineWidth(8.0f);
    }

    // Method to reset the surface if it flips vertical 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();

        // Create/recreate all lines to cater for change of orientation
        lineArray = new Line[NUM_LINES];
        for (int loop = 0; loop < NUM_LINES; ++loop)
        {
            lineArray[loop] = new Line();
        }
    }

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

        // Enable the vertex array client state (i.e. tell OpenGL ES that we're using vertices!)
        gl.glEnableClientState(GL_VERTEX_ARRAY);

        // Draw then update all particles in our array
        for (Line l : lineArray)
        {
            l.draw(gl);
            l.update();
        }

        // Disable the vertex array client state before leaving
        gl.glDisableClientState(GL_VERTEX_ARRAY);
    }

} // End of MyRenderer class
```

With all that done, you should have a number of bouncing blue lines displayed which looks something like this:



Student Project – MultiColoured Particle Fountain

Either take the code from project 3 (the particle fountain) or simply modify it so that:

- Particles now have three additional **float** properties for their **red, green** and **blue** colour values,
- When a particle is created, it gets assigned a random colour (i.e. each of the red/green/blue properties gets assigned a random value between 0.0f and 1.0f)
- When you draw a particle, make sure you use the **red/green/blue** properties in the **glColor4f** call which occurs before **glDrawArrays**. i.e. **`gl.glColor4f(red, green, blue, 1.0f);`**
- When a particle is reset (like when it goes off the screen), the colour gets re-randomised.

Easy as =D

Also, make the particles "fade out" by keeping an **alpha** property (float) which starts at **1.0f** and decreases by a small amount in the **update** method (perhaps 0.002f or so) each time the particle is drawn.

For this to work correctly, you'll need to enable blending and specify the blend function - so in the **onSurfaceCreated()** method of the renderer, add the lines:

```
gl.glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
gl.glEnable(GL_BLEND);
```

Modify you colour setting line to be **gl.glColor4f(red, green, blue, alpha);** so that we're actually using the alpha property we just created – and don't forget that you need to **reset the particle's alpha value to 1.0f** when you reset the particle!