Laboratory 2
2D Rendering

OpenGL Architecture

We will first take a brief look at the graphics pipeline of the OpenGL ES 1.x architecture.

The term *pipeline* is commonly used to illustrate how a tightly bound sequence of events relate to each other, as illustrated in Figure 1. In the case of OpenGL ES, the process accepts a series of numbers in one end and outputs the rendered image at the other end.

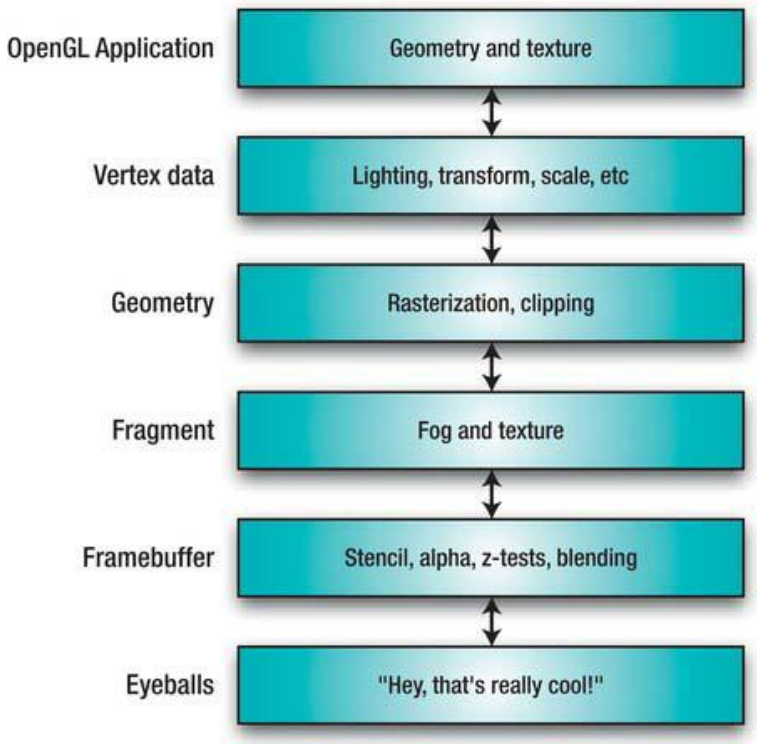

Figure 1: Basic overview of the OpenGL ES 1.x pipeline

- The first step is to take the data that describes some geometry along with information on how to handle lighting, colors, materials, and textures and send it into the pipeline.
- Next the data is moved and rotated, after which lighting on each object is calculated and stored. The scene must then be moved, rotated, and scaled based on the viewpoint you have set up. The viewpoint takes the form of a frustrum, a rectangular cone, which limits the scene to, ideally, a manageable level.
- Next the scene is clipped, meaning that only things that are likely to be visible are actually processed. All of the other things are culled out as early as possible and discarded. Much

of the history of real-time graphics development has to do with object culling techniques, some of which are very complex. Of course, if you can pre-cull objects on your own before submitting to the pipeline, so much the better. Perhaps the easiest is to simply tell whether an object is behind you making it completely skippable. Culling can also take place if the object is just too far away to see or is completely obscured by other objects.

- The remaining objects are now *projected* against the *viewport*, a virtual display.
- At this point is where *rasterization* takes place. Rasterization breaks apart the image into *fragments* that are in effect single pixels.
- Now the fragments can have texture and fog effects applied to them. Additional culling can likewise take place if the fog might obscure the more distant fragments, for example.
- The final phase is where the surviving fragments are written to the frame buffer, but only if they satisfy some last-minute operations. Here is where the fragment's alpha values are applied for translucency, along with depth tests to ensure that the closest fragments are drawn in front of further ones and stencil tests used to render to nonrectangular viewports.

The OpenGL ES 1.1 Reference Pages are given at

https://www.khronos.org/opengles/sdk/1.1/docs/man/.

The `Square` Class

Like in the first laboratory, create an Android project named `BouncySquare` in the `cg.bouncysquare` package, with an empty activity `BouncySquareActivity`, and no xml layout generated.

The first class we should add to our project is the `Square` class. First of all, we should add two imports to our class, namely

```
import javax.microedition.khronos.opengles.GL10;
import javax.microedition.khronos.opengles.GL11;
```

Java hosts several different OpenGL interfaces. The parent class is merely called `GL`, while OpenGL ES 1.0 uses `GL10`, and version 1.1 is imported as `GL11`, as shown above. You can also gain access to some extensions if your graphics hardware supports them via the `GL10Ext` package, supplied by the `GL11ExtensionPack`. The later versions are merely subclasses of the earlier ones; however, there are still some calls that are defined as taking only `GL10` objects, but those work if you cast the objects properly.

The `Square` class will consist of a constructor and a `draw` method.

The Constuctor

The first thing we need to do in the constructor is to define the vertices. You will rarely if ever do it this way because many objects could have thousands of vertices. In those cases, you would likely import them from any number of 3D file formats such as Imagination Technologies' POD files, 3D Studio's `.3ds` files, and so on. Here, since we are describing a 2D square, it is necessary to specify only `x` and `y` coordinates.

```
float vertices[] =
{
    -1.0f, -1.0f,
    1.0f, -1.0f,
    -1.0f, 1.0f,
    1.0f, 1.0f
};
```

And as you can see, the square is two units on a side.

Colors are defined similarly, but in this case, there are four components for each color: red, green, blue, and alpha (transparency). These map directly to the four vertices shown earlier, so the first color goes with the first vertex, and so on. You can use floats or a fixed or byte representation of the colors, with the latter saving a lot of memory if you are importing a very large model. Since we are using bytes, the color values go from 0 to 255. That means the first color sets red to 0, green to 0, blue to 0, and alpha to 255, yielding black. If you use floats or fixed point, they ultimately are converted to byte values internally.

```
byte maxColor=(byte)255;
byte colors[] =
{
    0, 0, 0, maxColor,
    maxColor, 0, 0, maxColor,
    0, 0, 0, maxColor,
    maxColor, 0, 0, maxColor,
};
```

Unlike its big desktop brother, which can render four-sided objects, OpenGL ES is limited to triangles only. Next, the connectivity array is created. This matches up the vertices to specific triangles. The first triplet says that vertices 0, 3, and 1 make up triangle 0, while the second triangle is comprised of vertices 0, 2, and 3.

```
byte indices[] =
{
    0, 3, 1,
    0, 2, 3
};
```

Once the colors, vertices, and connectivity array have been created, we may have to convert the internal Java formats of the values given above to those that OpenGL can understand, as shown below. This mainly ensures that the ordering of the bytes is right; otherwise, depending on the hardware, they might be in reverse order.

```
ByteBuffer vbb = ByteBuffer.allocateDirect(vertices.length * 4);
vbb.order(ByteOrder.nativeOrder());
mFVertexBuffer = vbb.asFloatBuffer();
mFVertexBuffer.put(vertices);
mFVertexBuffer.position(0);
mColorBuffer = ByteBuffer.allocateDirect(colors.length);
mColorBuffer.put(colors);
mColorBuffer.position(0);
mIndexBuffer = ByteBuffer.allocateDirect(indices.length);
mIndexBuffer.put(indices);
mIndexBuffer.position(0);
```

To this end, we need to add three private fields to the `Square` class

```
private FloatBuffer mFVertexBuffer;
private ByteBuffer mColorBuffer;
private ByteBuffer mIndexBuffer;
```

as well as the imports

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

The `draw` Method

The `draw(GL10 gl)` method is called by `SquareRenderer.drawFrame()`, discussed later.

First, we need to tell OpenGL how the vertices are ordering their faces. Vertex ordering can be critical when it comes to getting the best performance out of your software. It helps to have the ordering uniform across your model, which can indicate whether the triangles are facing toward or away from your viewpoint. The latter ones are called *backfacing triangles* the back side of your objects, so they can be ignored, cutting rendering time substantially. So, by specifying that the front face of the triangles are `GL_CW`, or clockwise, all counterclockwise triangles are culled. At the end of the method, they should reset to `GL_CCW`, which is the default.

```
gl.glFrontFace(GL11.GL_CW);
```

Next, pointers to the data buffers are handed over to the renderer. The call to `glVertexPointer()` specifies the number of elements per vertex (in this case two), that the data is floating point, and that the "stride" is 0 bytes. The data can be eight different formats, including floats, fixed, ints, short ints, and bytes. The latter three are available in both signed and unsigned variants. Stride is a useful way to let you interleave OpenGL data with your own as long as the data structures are constant. Stride is merely the number of bytes of user info packed between the GL data so the system can skip over it to the next bit it will understand.

```
gl.glVertexPointer(2, GL11.GL_FLOAT, 0, mFVertexBuffer);
```

The color buffer is sent across with a size of four elements, with the RGBA quadruplets using unsigned bytes (Java doesn't have unsigned types, but GL doesn't have to know), and it too has a stride equal to 0.

```
gl.glColorPointer(4, GL11.GL_UNSIGNED_BYTE, 0, mColorBuffer);
```

Now, the actual drawing takes place. This requires the connectivity array. The first parameter says what the format the geometry is in, in other words, triangles, triangle lists, points, or lines. In our case, we use `GL_TRIANGLES`. The second parameter specifies the number of elements to be rendered. Then comes the data type, and finally the connectivity array.

```
gl.glDrawElements(GL11.GL_TRIANGLES, 6, GL11.GL_UNSIGNED_BYTE, mIndexBuffer);
```

The SquareRenderer Class

Now our square needs a driver and a way to display itself on the screen. Create a new class called `SquareRenderer` which implements the `GLSurfaceView.Renderer` interface.

The EGL libraries bind the OpenGL drawing surface to the system but are buried within `GLSurfaceView` in this case, as shown in the imports below. EGL is primarily used for allocating and managing the drawing surfaces and is part of an OpenGL ES extension, so it is platform independent.

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

The Constuctor

In the constructor, the `Square` object is allocated and cached. Thus, we need to add a private field to the `SquareRenderer` class

```
private Square mSquare;
```

and initialize it in the constructor

```
mSquare = new Square();
```

The onDrawFrame method

The `onDrawFrame(GL10 gl)` method is the root refresh method; this constructs the image each time through, many times a second. And the first call is typically to clear the entire screen, as shown below. Considering that a frame can be constructed out of several components, you are given the option to select which of those should be cleared every frame. The color buffer holds all of the RGBA color data, while the depth buffer is used to ensure that the closer items properly obscure the further items.

```
gl.glClear(GL10.GL_COLOR_BUFFER_BIT | GL10.GL_DEPTH_BUFFER_BIT);
```

We now start dealing with the actual 3D parameters. All that is being done here is setting the values to ensure that the example geometry is immediately visible.

```
gl.glMatrixMode(GL10.GL_MODELVIEW);
gl.glLoadIdentity();
```

Next, we translate the square up and down. To get a nice, smooth motion, the actual translation value is based on a sine wave. The value of the private float field `mTransY` of the `SquareRenderer` class is simply used to generate a final up and down value that ranges from -1 to +1. Each time through `drawFrame`, the translation is increased by 0.075. Since we're taking the sine of this, it isn't necessary to loop the value back on itself, because sine will do that for us.

```
gl.glTranslatef(0.0f,(float)Math.sin(mTransY), -3.0f);
mTransY += 0.075f;
```

The following import should also be added:

```
import java.lang.Math;
```

We need to tell OpenGL to expect both vertex and color data:

```
gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
gl.glEnableClientState(GL10.GL_COLOR_ARRAY);
```

Finally, after all of this setup code, we can call the actual drawing routine of the `mSquare` that you have seen before, as shown below.

```
mSquare.draw(gl);
```

The `onSurfaceChanged` method

The `onSurfaceChanged(GL10 gl, int width, int height)`, is called whenever the screen changes size or is created at startup. Here it is also being used to set up the viewing *frustum*, which is the volume of space that defines what you can actually see. If any of your scene elements lay outside of the frustum, they are considered invisible so are clipped, or culled out, to prevent that further operations are done on them.

The `glViewport` merely permits you to specify the actual dimensions and placement of your OpenGL window. This will typically be the size of your main screen, with a location 0.

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

Next, we set the matrix mode. What this does is to set the current working matrix that will be acted upon when you make any general-purpose matrix management calls. In this case, we switch to the `GL_PROJECTION` matrix, which is the one that projects the 3D scene to your 2D screen. `glLoadIdentity` resets the matrix to its initial values to erase any previous settings.

```
gl.glMatrixMode(GL10.GL_PROJECTION);
gl.glLoadIdentity();
```

Now you can set the actual frustum using the aspect ratio and the six clipping planes: left, right, bottom, top, near, far, which are also the six parameters of the `glFrustumf` method. The `left`, `right` parameters specify the coordinates for the left and right vertical clipping planes, the `bottom`, `top` parameters specify the coordinates for the bottom and top horizontal clipping planes, and the `near`, `far` parameters specify the distances to the near and far depth clipping planes (both distances must be positive).

```
float ratio = (float) width / height;
gl.glFrustumf(-ratio, ratio, -1, 1, 1, 10);
```

The `onSurfaceCreated` Method

The last method of the `SquareRenderer` class, namely `onSurfaceCreated(GL10 gl, EGLConfig config)`is called upon surface creation and deals mainly with initialization steps.

We first ensure that any dithering is turned off, because it defaults to on. Dithering in OpenGL makes screens with limited color palettes look somewhat nicer but at the expense of performance of course.

```
gl.glDisable(GL10.GL_DITHER);
```

Next, `glHint` is used to tell OpenGL ES to do what it thinks best by accepting certain trade-offs: usually speed vs. quality. Other hintable settings include fog and various smoothing options.

```
gl.glHint(GL10.GL_PERSPECTIVE_CORRECTION_HINT, GL10.GL_FASTEST);
```

Another one of the many states we can set is the color that the background assumes when cleared. In this case, the background is black.

```
gl.glClearColor(0,0,0,0);
```

At last, we set some other modes. First line says to cull out faces (triangles) that are aimed away from us. The next line tells it to use smooth shading so the colors blend across the surface. The only other value is `GL_FLAT`, which, when activated, will display the face in the color of the last vertex drawn. And the last line enables depth testing, also known as *z-buffering*, discussed in later laboratories.

```
gl.glEnable(GL10.GL_CULL_FACE);
gl.glShadeModel(GL10.GL_SMOOTH);
gl.glEnable(GL10.GL_DEPTH_TEST);
```

The `BouncySquareActivity` Class

Finally, the activity file will need to be modified. First of all, the `BouncySquareActivity` class will extend the `Activity` class. The following imports need to be added to the `BouncySquareActivity` class:

```
import android.app.Activity;
import android.opengl.GLSurfaceView;
import android.os.Bundle;
import android.view.WindowManager;
```

The `onCreate` method

The only method that we need to override from the `Activity` class is the `onCreate(Bundle savedInstanceState)` method.

We first call the method from the base class:

```
super.onCreate(savedInstanceState);
```

Then, we set our activity to run full screen:

```
getWindow().setFlags(WindowManager.LayoutParams.FLAG_FULLSCREEN,
        WindowManager.LayoutParams.FLAG_FULLSCREEN);
```

Finally, the `GLSurfaceView` is actually allocated and bound to our custom renderer, `SquareRenderer`.

```
GLSurfaceView view = new GLSurfaceView(this);
view.setRenderer(new SquareRenderer());
setContentView(view);
```

Now compile and run. You should see something that looks like Figure 2.

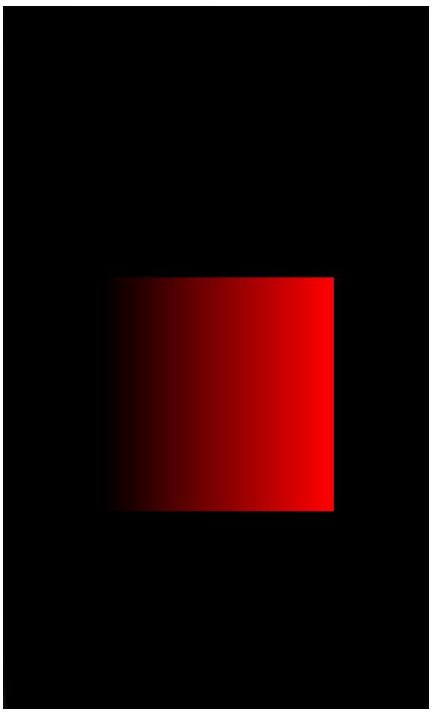

Figure 2: The bouncy square

Assignment

- Replace the first number in the vertices array with -2.0, instead of -1.0.
- Replace the color of the bouncy square with green, and then with blue.
- In the `glDrawElements` call, use the other symbolic constants `GL_POINTS`, `GL_LINE_STRIP`, `GL_LINE_LOOP`, `GL_LINES`, `GL_TRIANGLE_STRIP`, `GL_TRIANGLE_FAN`.
- Increase the value of `mTransY` with increments of 0.3.
- Leave the dithering enabled.
- Change the color (and alpha value) of the background.