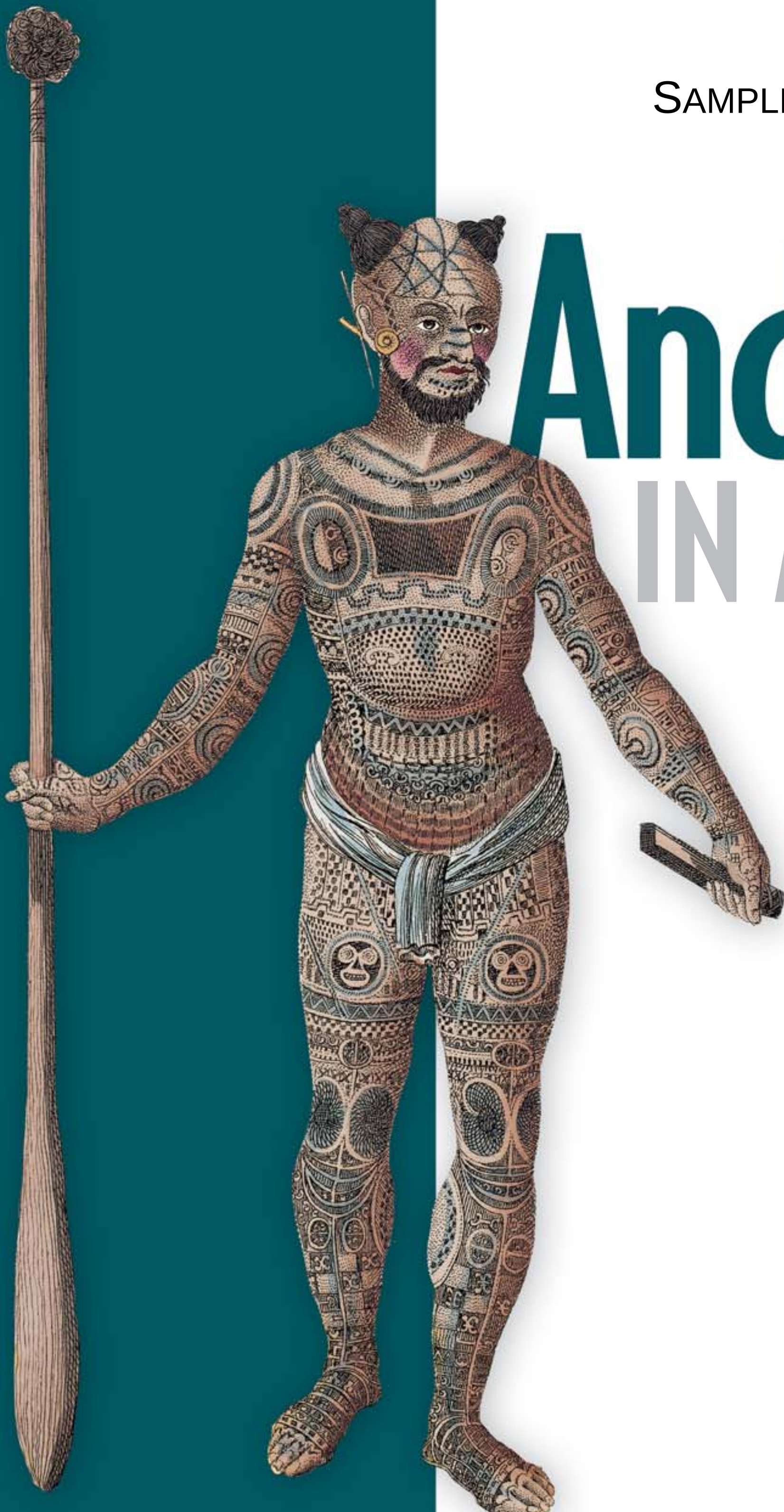
SAMPLE CHAPTER



Android IN ACTION

THIRD EDITION

W. Frank Ableson
Robi Sen
Chris King
C. Enrique Ortiz

MANNING

Android in Action, Third Edition

by W. Frank Abelson
Robi Sen
Chris King
C. Enrique Ortiz

Chapter 9

brief contents

9 Graphics and animation

This chapter covers

- Drawing graphics in Android
- Applying the basics of OpenGL for embedded systems (ES)
- Animating with Android

By now, you should've picked up on the fact that it's much easier to develop Android applications than it is to use other mobile application platforms. This ease of use is especially apparent when you're creating visually appealing UIs and metaphors, but there's a limit to what you can do with typical Android UI elements (such as those we discussed in chapter 3). In this chapter, we'll look at how to create graphics using Android's Graphics API, discuss how to develop animations, and explore Android's support for the OpenGL standard, as well as introduce you to Android's new cross-platform, high-performance graphics language RenderScript. (To see examples of what you can do with Android's graphics platform, go to www.omnigsoft.com/Android/ADC/readme.html.)

First, we're going to show you how to draw simple shapes using the Android 2D Graphics API, using Java and then XML to describe 2D shapes. Next, we'll look at making simple animations using Java and the Graphics API to move pixels around, and then using XML to perform a frame-by-frame animation. After that we'll examine Android's support of the OpenGL ES API, make a simple shape, and then make a more complex, rotating, three-dimensional shape. Finally we'll introduce RenderScript, a low-level, C-derived, native language that allows developers to take advantage of multicore systems and graphics accelerators to make more performant, visually intensive applications.

If you've ever worked with graphics in Java, you'll likely find the Graphics API and how graphics work in Android familiar. If you've worked with OpenGL, you'll find Android's implementation of OpenGL ES reasonably straightforward. You must remember, though, that cell phones, tablets, and other mobile devices don't have the graphical processing power of a desktop. Regardless of your experience, you'll find the Android Graphics API both powerful and rich, allowing you to accomplish even some of the most complex graphical tasks.

> **NOTE** You can find more information on the differences between OpenGL and OpenGL ES to help you determine the level of effort in porting code at the Khronos website. For example, the OpenGL ES 1.5 specification at http://mng.bz/qapb provides information on differences between OpenGL and OpenGL ES.

9.1 Drawing graphics in Android

In this section, we'll cover Android's graphical capabilities and show you examples of how to make simple 2D shapes. We'll be applying the `android.graphics` package (see http://mng.bz/CIFJ), which provides all the low-level classes you need to create graphics. The graphics package supports such things as bitmaps (which hold pixels), canvases (what your draw calls draw on), primitives (such as rectangles and text), and paints (which you use to add color and styling). Although these aren't the only graphics packages, they're the main ones you'll use in most applications. Generally, you use Java to call the Graphics API to create complex graphics.

To demonstrate the basics of drawing a shape with Java and the Graphics API, let's look at a simple example in the following listing, where we'll draw a rectangle.

Listing 9.1 simpleshape.java

```
package com.msi.manning.chapter9.SimpleShape;
public class SimpleShape extends Activity {
@Override
protected void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    setContentView(new SimpleView(this));
}
private static class SimpleView extends View {
    private ShapeDrawable mDrawable =
        new ShapeDrawable();                      // 1 Create new ShapeDrawable to hold Drawable
    public SimpleView(Context context) {          // 2 Set up View
        super(context);
        setFocusable(true);
        this.mDrawable =
            new ShapeDrawable(new RectShape());   // 3 Create Rectangle, assign to mDrawable
        this.mDrawable.getPaint().setColor(0xFFFF0000);
```

```
        }
        @Override
        protected void onDraw(Canvas canvas) {
            int x = 10;
            int y = 10;
            int width = 300;
            int height = 50;
            this.mDrawable.setBounds(x, y, x + width, y + height);
            this.mDrawable.draw(canvas);
            y += height + 5;
        }
    }
}
```

First, we need to import the necessary packages, including graphics. Then we import `ShapeDrawable`, which will support adding shapes to our drawing, and then shapes, which supports several generic shapes (including `RectShape`) that we'll use. Next, we need to create ❶ and then set up a `View` ❷. After this, we create a new `ShapeDrawable` to add our `Drawable` to ❸. After we have a `ShapeDrawable`, we can assign shapes to it. In the code, we use the `RectShape`, but we could've used `OvalShape`, `PathShape`, `RectShape`, `RoundRectShape`, or `Shape`. We then use the `onDraw()` method to draw the `Drawable` on the `Canvas`. Finally, we use the `Drawable`'s `setBounds()` method to set the boundary (a rectangle) in which we'll draw our rectangle using the `draw()` method.

When you run listing 9.1, you should see a simple rectangle like the one shown in figure 9.1 (it's red, although you can't see the color in the printed book).

Another way to do the same thing is through XML. Android allows you to define shapes to draw in an XML resource file.

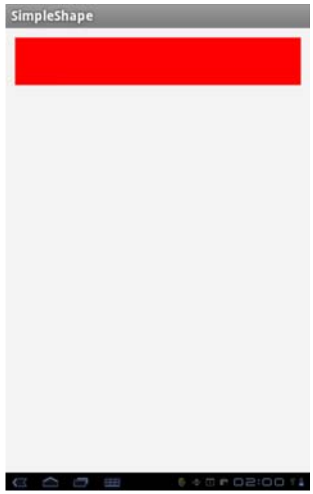


Figure 9.1 A simple rectangle drawn using Android's Graphics API

9.1.1 Drawing with XML

With Android, you can create simple drawings using an XML file approach. You might want to use XML for several reasons. One basic reason is because it's simple to do. Also, it's worth keeping in mind that graphics described by XML can be programmatically changed later, so XML provides a simple way to do initial design that isn't necessarily static.

To create a drawing with XML, create one or more `Drawable` objects, which are defined as XML files in your drawable directory, such as res/drawable. The XML to create a simple rectangle looks like this:

```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:android="http://schemas.android.com/apk/res/android">
```

```
    <solid android:color="#FF0000FF"/>
</shape>
```

With Android XML drawable shapes, the default is a rectangle, but you can choose a different shape by using the `type` tag and selecting the value `oval`, `rectangle`, `line`, or `arc`. To use your XML shape, you need to reference it in a layout, as shown in listing 9.2. The layout resides in res/layout.

ARGB color values

Android uses of Alpha, Red, Green, Blue (ARGB) values common in the software industry for defining color values throughout the Android API. In RGB, colors are defined as ints made up of four bytes: red, green, and blue, plus an alpha. Each value can be a number from 0 to 255 that is converted to hexadecimal (hex). The alpha indicates the level of transparency from 0 to 255.

For example, to create a transparent yellow, we might start with an alpha of 50.2% transparency, where the hex value is 0x80: this is 128, which is 50.2% of 255. To get yellow, we need red plus green. The number 255 in hex for red and green is FF. No blue is needed, so its value is 00. Thus a transparent yellow is 80FFFF00. This may seem confusing, but numerous ARGB color charts are available that show the hexadecimal values of a multitude of colors.

Listing 9.2 xmllayout.xml

```
<?xml version="1.0" encoding="utf-8"?>
<ScrollView xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content">
     <LinearLayout
        android:orientation="vertical"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content">
     <ImageView android:layout_width="fill_parent"
        android:layout_height="50dip"
        android:src="@drawable/simplerectangle" />
     </LinearLayout>
</ScrollView>
```

All you need to do is create a simple `Activity` and place the UI in a `ContentView`, as follows:

```
public class XMLDraw extends Activity {
    @Override
    public void onCreate(Bundle icicle) {
        super.onCreate(icicle);
        setContentView(R.layout.xmldrawable);
    }
}
```

If you run this code, it draws a simple rectangle. You can make more complex drawings or shapes by stacking or ordering XML drawables, and you can include as many shapes as you want or need, depending on space. Let's explore what multiple shapes might look like next.

9.1.2 Exploring XML drawable shapes

One way to draw multiple shapes with XML is to create multiple XML files that represent different shapes. A simple way to do this is to change the xmldrawable.xml file to look like the following listing, which adds a number of shapes and stacks them vertically.

Listing 9.3 xmldrawable.xml

```
<?xml version="1.0" encoding="utf-8"?>
<ScrollView xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content">
    <LinearLayout
        android:orientation="vertical"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content">
    <ImageView android:layout_width="fill_parent"
        android:layout_height="50dip"
        android:src="@drawable/shape_1" />
    <ImageView android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:src="@drawable/shape_2" />
    <ImageView
        android:layout_width="fill_parent"
        android:layout_height="50dip"
        android:src="@drawable/shape_3" />
    <ImageView
        android:layout_width="fill_parent"
        android:layout_height="50dip"
        android:src="@drawable/shape_4" />
    </LinearLayout>
</ScrollView>
```

Try adding any of the shapes shown in the following code snippets into the res/drawable folder. You can sequentially name the files shape_*n*.xml, where *n* is some number. Or you can give the files any acceptable name as long as the XML file defining the shape is referenced in the xmldrawable.xml file.

In the following code, we're creating a rectangle with rounded corners. We've added a tag called `padding`, which allows us to define padding or space between the object and other objects in the UI:

```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:android="http://schemas.android.com/apk/res/android"
type="oval" >
        <solid android:color="#00000000"/>
        <padding android:left="10sp" android:top="4sp"
```

```
        android:right="10sp" android:bottom="4sp" />
        <stroke android:width="1dp" android:color="#FFFFFFFF"/>
</shape>
```

We're also using the `stroke` tag, which allows us to define the style of the line that makes up the border of the oval, as shown here:

```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:android="http://schemas.android.com/apk/res/android">
    <solid android:color="#FF0000FF"/>
    <stroke android:width="4dp" android:color="#FFFFFFFF"
        android:dashWidth="1dp" android:dashGap="2dp" />
    <padding android:left="7dp" android:top="7dp"
        android:right="7dp" android:bottom="7dp" />
    <corners android:radius="4dp" />
</shape>
```

The next snippet introduces the `corners` tag, which allows us to make rounded corners with the attribute `android:radius`:

```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:android="http://schemas.android.com/apk/res/android"
type="oval">
    <gradient android:startColor="#FFFF0000" android:endColor="#80FF00FF"
        android:angle="270"/>
    <padding android:left="7dp" android:top="7dp"
        android:right="7dp" android:bottom="7dp" />
    <corners android:radius="8dp" />
</shape>
```

Finally, we create a shape of the type `line` with a `size` tag using the `android:height` attribute, which allows us to describe the number of pixels used on the vertical to size the line:

```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:android="http://schemas.android.com/apk/res/android"
type="line" >
    <solid android:color="#FFFFFFFF"/>
    <stroke android:width="1dp" android:color="#FFFFFFFF"
        android:dashWidth="1dp" android:dashGap="2dp" />
    <padding android:left="1dp" android:top="25dp"
        android:right="1dp" android:bottom="25dp" />
    <size android:height="23dp" />
</shape>
```

If you run this code, you should see something like figure 9.2.

As you can see, Android provides the ability for developers to programmatically draw anything they need. In the next section, we'll look at what you can draw with Android's animation capabilities.

9.2 Creating animations with Android's Graphics API

If a picture says a thousand words, then an animation must speak volumes. Android supports multiple methods of creating animation, including through XML, as you saw

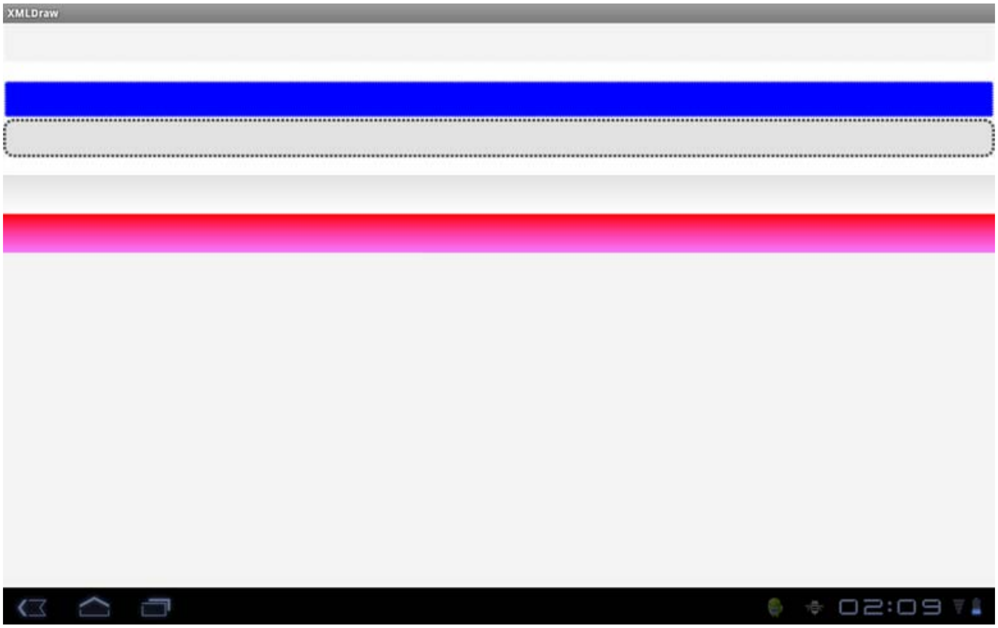


Figure 9.2 **Various shapes drawn using XML**

in chapter 3; via Android's XML frame-by-frame animations using the Android Graphics API; and via Android's support for OpenGL ES. In this section, you'll create a simple animation of a bouncing ball using Android's frame-by-frame animation.

9.2.1 *Android's frame-by-frame animation*

Android allows you to create simple animations by showing a set of images one after another to give the illusion of movement, much like stop-motion film. Android sets each frame image as a drawable resource; the images are then shown one after the other in the background of a `View`. To use this feature, you define a set of resources in an XML file and then call `AnimationDrawable.start()`.

To demonstrate this method for creating an animation, you need to download this project from the Manning website (www.manning.com/ableson3) so you'll have the images. The images for this exercise are six representations of a ball bouncing. Next, create a project called XMLanimation, and create a new directory called /anim under the /res resources directory. Place all the images for this example in res/drawable. Then, create an XML file called Simple_animation.xml that contains the code shown in the following listing.

Listing 9.4 Simple_animation.xml

```
<?xml version="1.0" encoding="utf-8"?>
 <animation-list xmlns:android="http://schemas.android.com/apk/res/android"
 id="selected" android:oneshot="false">
```

```
  <item android:drawable="@drawable/ball1" android:duration="50" />
  <item android:drawable="@drawable/ball2" android:duration="50" />
  <item android:drawable="@drawable/ball3" android:duration="50" />
  <item android:drawable="@drawable/ball4" android:duration="50" />
  <item android:drawable="@drawable/ball5" android:duration="50" />
  <item android:drawable="@drawable/ball6" android:duration="50" />
 </animation-list>
```

The XML file defines the list of images to be displayed for the animation. The XML `<animation-list>` tag contains the tags for two attributes: `drawable`, which describes the path to the image, and `duration`, which describes the length of time to show the image, in nanoseconds.

Now, edit the main.xml file to look like the following listing.

Listing 9.5 main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
      <ImageView android:id="@+id/simple_anim"
          android:layout_width="wrap_content"
          android:layout_height="wrap_content"
          android:gravity="center"
          android:layout_centerHorizontal="true"
           />
<TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Hello World, XMLAnimation"
    />
</LinearLayout>
```

All we've done to the file is added an `ImageView` tag that sets up the layout for the `ImageView`. Finally, create the code to run the animation, as follows.

Listing 9.6 xmlanimation.java

```
public class XMLAnimation extends Activity
{
    @Override
    public void onCreate(Bundle icicle) {
        super.onCreate(icicle);
        setContentView(R.layout.main);
        ImageView img =
        (ImageView)findViewById(R.id.simple_anim);
        img. setBackgroundResource(R.anim.simple_animation);
        MyAnimationRoutine mar =
        new MyAnimationRoutine();
        MyAnimationRoutine2 mar2 =
        new MyAnimationRoutine2();
        Timer t = new Timer(false);
```

Bind resources to ImageView

Call subclasses to start and stop animation

```
        t.schedule(mar, 100);
        Timer t2 = new Timer(false);
        t2.schedule(mar2, 5000);
    }
    class MyAnimationRoutine extends TimerTask {
        @Override
        public void run() {
            ImageView img = (ImageView) findViewById(R.id.simple_anim);
            AnimationDrawable frameAnimation = (AnimationDrawable)
           img.getBackground();
            frameAnimation.start();
        }
    }
    class MyAnimationRoutine2 extends TimerTask {
        @Override
        public void run() {
            ImageView img = (ImageView) findViewById(R.id.simple_anim);
            AnimationDrawable frameAnimation = (AnimationDrawable)
           img.getBackground();
            frameAnimation.stop();
        }
    }
}
```

1 Allow wait time before starting animation

Listing 9.6 may be slightly confusing because we've used the `TimerTask` classes. Because we can't control the animation from within the `OnCreate()` method, we need to create two such subclasses to call `AnimationDrawable`'s `start()` and `stop()` methods, respectively. The first subclass, `MyAnimationRoutine`, extends `TimerTask` **1** and calls the `frameAnimation.start()` method for the `AnimationDrawable` bound to the `ImageView` background. If you run the project now, you should see something like figure 9.3.

As you can see, creating an `Animation` with XML in Android is pretty simple. You can make animations that are reasonably complex, as you would with any stop-motion-type movie; but to create more sophisticated animations programmatically, you need to use Android's 2D and 3D graphics abilities. In the next section, we'll show you how to do just that.

9.2.2 Programmatically creating an animation

In the previous section, you used Android's frame-by-frame animation capabilities to show a series of images in a loop that gives the impression of movement. In this section, you'll programmatically animate a globe so that it moves around the screen.

To create this animation, you'll animate a graphics file (a PNG file) with a ball that appears to be bouncing around inside the Android viewing window. You'll create a `Thread` in which the animation will run and a `Handler` that will help communicate back to the program messages that reflect the changes in the state of the animation. You'll use this same approach in section 9.3 when we talk about OpenGL ES. You'll find that this approach is useful for creating most complex graphics applications and animations.

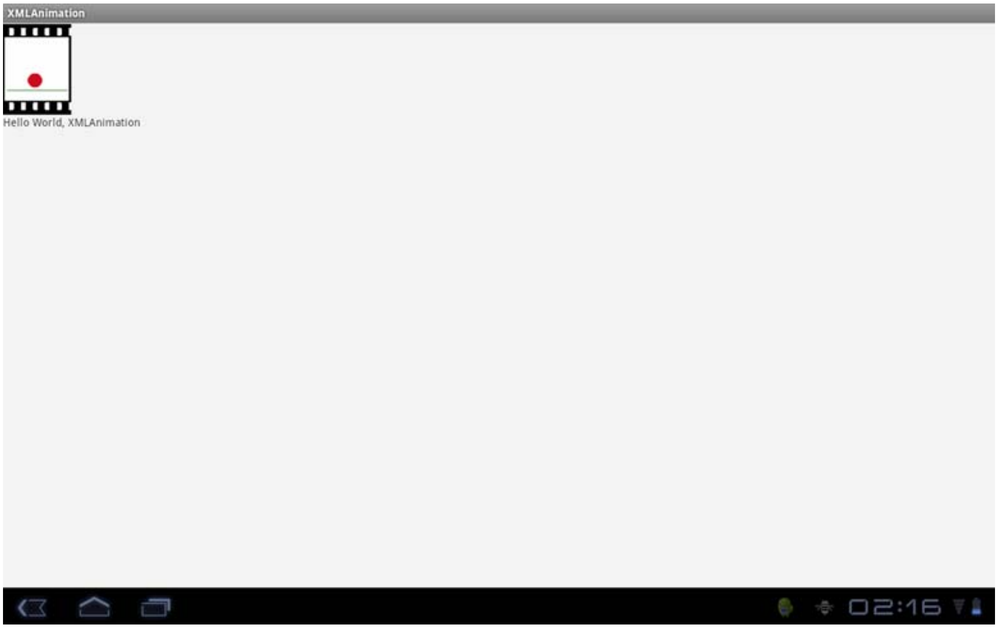


Figure 9.3 **Making a ball bounce using an Android XML animation**

CREATING THE PROJECT

This example's animation technique uses an image bound to a sprite. In general, *sprite* refers to a two-dimensional image or animation that is overlaid onto a background or more complex graphical display. For this example, you'll move the sprite around the screen to give the appearance of a bouncing ball. To get started, create a new project called BouncingBall with a `BounceActivity`. You can copy and paste the code in the following listing for the BounceActivity.java file.

Listing 9.7 BounceActivity.java

```
public class BounceActivity extends Activity {
     protected static final int GUIUPDATEIDENTIFIER = 0x101;
     Thread myRefreshThread = null;
     BounceView myBounceView = null;
     Handler myGUIUpdateHandler = new Handler() {
          public void handleMessage(Message msg) {
               switch (msg.what) {
                    case BounceActivity.GUIUPDATEIDENTIFIER:
                         myBounceView.invalidate();
                         break;
               }
               super.handleMessage(msg);
          }
     };
     @Override
     public void onCreate(Bundle icicle) {
```

1 Create unique identifier

2 Create handler

```
            super.onCreate(icicle);
            this.requestWindowFeature(Window.FEATURE_NO_TITLE);
            this.myBounceView = new BounceView(this);                  ❸ Create view
            this.setContentView(this.myBounceView);
            new Thread(new RefreshRunner()).start();
      }
      class RefreshRunner implements Runnable {
            public void run() {                                       ❹ Run animation
                  while (!Thread.currentThread().isInterrupted()) {
                        Message message = new Message();
                        message.what = BounceActivity.GUIUPDATEIDENTIFIER;
                        BounceActivity.this.myGUIUpdateHandler
.sendMessage(message);
                        try {
                              Thread.sleep(100);
                        } catch (InterruptedException e) {
                              Thread.currentThread().interrupt();
                        }
                  }
            }
      }
}
```

First we import the `Handler` and `Message` classes, and then we create a unique identifier to allow us to send a message back to our program to update the `View` in the main thread. We need to send a message telling the main thread to update the `View` each time the child thread has finished drawing the ball. Because different messages can be thrown by the system, we need to guarantee the uniqueness of our message to our handler by creating a unique identifier called `GUIUPDATEIDENTIFIER` ❶. Next, we create the `Handler` that will process our messages to update the main `View` ❷. A `Handler` allows us to send and process `Message` classes and `Runnable` objects associated with a thread's message queue.

`Handlers` are associated with a single thread and its message queue, but their methods can be called from any thread. Thus we can use the `Handler` to allow objects running in another thread to communicate changes in state back to the thread that spawned them, or vice versa.

> **NOTE** For more information about handling long-running requests in your applications, see http://mng.bz/K0H4.

We set up a `View` ❸ and create the new thread. Finally, we create a `RefreshRunner` inner class implementing `Runnable` that will run unless something interrupts the thread, at which point a message is sent to the `Handler` to call `BounceView`'s `invalidate()` method ❹. The `invalidate()` method invalidates the `View` and forces a refresh.

You've got your new project. Now you need to create the code that will perform the animation and create a `View`.

MAKING ANIMATION HAPPEN

The example uses an image of a globe, which you can obtain from www.manning.com/ableson3. (Alternatively, you can use any PNG file you'd like.) You'll also have the Android logo as a background; it's included with the source code downloads. Make sure to drop the images into res/drawable/.

Next, create a Java file called BounceView, using the code from the following listing.

Listing 9.8 BounceView.java

```
public class BounceView extends View {
     protected Drawable mySprite;
     protected Point mySpritePos = new Point(0,0);
     protected enum HorizontalDirection {LEFT, RIGHT} ;
     protected enum VerticalDirection {UP, DOWN} ;
     protected HorizontalDirection myXDirection =
HorizontalDirection.RIGHT;
     protected VerticalDirection myYDirection = VerticalDirection.UP;
     public BounceView(Context context) {
          super(context);
this.setBackground(this.getResources().getDrawable(R.drawable.android));
this.mySprite =
     this.getResources().getDrawable(R.drawable.world);
     }
     @Override
     protected void onDraw(Canvas canvas) {
this.mySprite.setBounds(this.mySpritePos.x,
     this.mySpritePos.y,
     this.mySpritePos.x + 50, this.mySpritePos.y + 50);
          if (mySpritePos.x >= this.getWidth() -
mySprite.getBounds().width()) {
               this.myXDirection = HorizontalDirection.LEFT;
          } else if (mySpritePos.x <= 0) {
               this.myXDirection = HorizontalDirection.RIGHT;
          }
          if (mySpritePos.y >= this.getHeight() -
mySprite.getBounds().height()) {
               this.myYDirection = VerticalDirection.UP;
          } else if (mySpritePos.y <= 0) {
               this.myYDirection = VerticalDirection.DOWN;
          }
          if (this.myXDirection ==
HorizontalDirection.RIGHT) {
               this.mySpritePos.x += 10;
          } else {
               this.mySpritePos.x -= 10;
          }
          if (this.myYDirection ==
                         VerticalDirection.DOWN) {
               this.mySpritePos.y += 10;
          } else {
               this.mySpritePos.y -= 10;
          }
          this.mySprite.draw(canvas);
     }
}
```

1 Get image file and map to sprite

2 Set bounds of globe

3 Move ball left or right, up or down

4 Check if ball is trying to leave screen

In this listing, we do all the real work of animating the image. First, we create a `Drawable` to hold the globe image and a `Point` that we use to position and track the globe as we animate it. Next, we create enumerations (`enums`) to hold directional values for horizontal and vertical directions, which we'll use to keep track of the moving globe. Then we map the globe to the `mySprite` variable and set the Android logo as the background for the animation ❶.

Now that we've done the setup work, we create a new `View` and set all the boundaries for the `Drawable` ❷. After that, we create simple conditional logic that detects whether the globe is trying to leave the screen; if it starts to leave the screen, we change its direction ❸. Then we provide simple conditional logic to keep the ball moving in the same direction if it hasn't encountered the bounds of the `View` ❹. Finally, we draw the globe using the `draw()` method.

If you compile and run the project, you should see the globe bouncing around in front of the Android logo, as shown in figure 9.4.

Although this animation isn't too exciting, you could—with a little extra work—use the key concepts (dealing with boundaries, moving `drawables`, detecting changes, dealing with threads, and so on) to create something like the Google Lunar Lander example game or even a simple version of Asteroids. If you want more graphics power and want to easily work with 3D objects to create things such as games or sophisticated animations, you'll learn how in the next section on OpenGL ES.

Figure 9.4 Animation of a globe bouncing in front of the Android logo

9.3 Introducing OpenGL for Embedded Systems

One of the most interesting features of the Android platform is its support of *OpenGL for Embedded Systems* (*OpenGL ES*). OpenGL ES is the embedded systems version of the popular OpenGL standard, which defines a cross-platform and cross-language API for computer graphics. The OpenGL ES API doesn't support the full OpenGL API, and much of the OpenGL API has been stripped out to allow OpenGL ES to run on a variety of mobile phones, PDAs, video game consoles, and other embedded systems. OpenGL ES was originally developed by the Khronos Group, an industry consortium. You can find the most current version of the standard at www.khronos.org/opengles/.

OpenGL ES is a fantastic API for 2D and 3D graphics, especially for graphically intensive applications such as games, graphical simulations, visualizations, and all sorts of animations. Because Android also supports 3D hardware acceleration, developers can make graphically intensive applications that target hardware with 3D accelerators.

Android 2.1 supports the OpenGL ES 1.0 standard, which is almost equivalent to the OpenGL 1.3 standard. If an application can run on a computer using OpenGL 1.3, it should be possible to run it on Android after light modification, but you need to consider the hardware specifications of your Android handset. Although Android offers support for hardware acceleration, some handsets and devices running Android have had performance issues with OpenGL ES in the past. Before you embark on a project using OpenGL, consider the hardware you're targeting and do extensive testing to make sure that you don't overwhelm your hardware with OpenGL graphics.

Because OpenGL and OpenGL ES are such broad topics, with entire books dedicated to them, we'll cover only the basics of working with OpenGL ES and Android. For a much deeper exploration of OpenGL ES, check out the specification and the OpenGL ES tutorial at http://mng.bz/0tdm. After reading this section on Android support for OpenGL ES, you should have enough information to follow a more in-depth discussion of OpenGL ES, and you should be able to port your code from other languages (such as the tutorial examples) into the Android framework. If you already know OpenGL or OpenGL ES, then the OpenGL commands will be familiar; concentrate on the specifics of working with OpenGL on Android.

> **NOTE** For another good OpenGL resource from Silicon Graphics see www.glprogramming.com/red/index.html.

9.3.1 *Creating an OpenGL context*

Keeping in mind the comments we made in the introduction to this section, let's apply the basics of OpenGL ES to create an `OpenGLContext` and a `Window` to draw in. Much of this task will seem overly complex compared to Android's Graphics API. The good news is that you have to do this setup work only once.

> **NOTE** Much of the material covered here will require further detailed explanation if you aren't already experienced with OpenGL. For more information, we suggest that you refer directly to the documentation from OpenGL at www.opengl.org/.

You'll use the general processes outlined in the following sections to work with OpenGL ES in Android:

1. Create a custom `View` subclass.
2. Get a handle to an `OpenGLContext`, which provides access to Android's OpenGL ES functionality.
3. In the `View`'s `onDraw()` method, use the handle to the GL object and then use its methods to perform any GL functions.

Following these basic steps, first you'll create a class that uses Android to create a blank surface to draw on. In section 9.3.2, you'll use OpenGL ES commands to draw a square and an animated cube on the surface. To start, open a new project called OpenGLSquare and create an `Activity` called `OpenGLSquare`, as shown in the following listing.

Listing 9.9 OpenGLSquare.java

```
public class SquareActivity extends Activity {
    @Override
    public void onCreate(Bundle icicle) {
        super.onCreate(icicle);
        setContentView(new DrawingSurfaceView(this));
    }
    class DrawingSurfaceView extends SurfaceView implements
    SurfaceHolder.Callback  {
        public SurfaceHolder mHolder;
        public DrawingThread mThread;
        public DrawingSurfaceView(Context c) {
            super(c);
            init();
        }
        public void init() {
            mHolder = getHolder();
            mHolder.addCallback(this);
            mHolder.setType(SurfaceHolder.SURFACE_TYPE_GPU);
        }
        public void surfaceCreated(SurfaceHolder holder) {
            mThread = new DrawingThread();
            mThread.start();
        }
        public void surfaceDestroyed(SurfaceHolder holder) {
            mThread.waitForExit();
            mThread = null;
        }
        public void surfaceChanged(SurfaceHolder holder,
        int format, int w, int h) {
            mThread.onWindowResize(w, h);
        }
        class DrawingThread extends Thread {
            boolean stop;
            int w;
            int h;
            boolean changed = true;
            DrawingThread() {
                super();
                stop = false;
                w = 0;
                h = 0;
            }
            @Override
            public void run() {
 EGL10 egl = (EGL10)EGLContext.getEGL();
                EGLDisplay dpy =
                egl.eglGetDisplay(EGL10.EGL_DEFAULT_DISPLAY);
                int[] version = new int[2];
                egl.eglInitialize(dpy, version);
                int[] configSpec = {
                   EGL10.EGL_RED_SIZE,       5,
                   EGL10.EGL_GREEN_SIZE,     6,
                   EGL10.EGL_BLUE_SIZE,      5,
```

1 Handle creation and destruction

2 Do drawing

3 Register as callback

4 Create thread to do drawing

5 Get EGL Instance

6 Specify configuration to use

```
                EGL10.EGL_DEPTH_SIZE,   16,
                EGL10.EGL_NONE
            };
            EGLConfig[] configs = new EGLConfig[1];
            int[] num_config = new int[1];
            egl.eglChooseConfig(dpy, configSpec, configs, 1,
 num_config);
           EGLConfig config = configs[0];
            EGLContext context = egl.eglCreateContext(dpy,
 config, EGL10.EGL_NO_CONTEXT, null);
           EGLSurface surface = null;
           GL10 gl = null;
           while(!stop) {
               int W, H;
           boolean updated;
               synchronized(this) {
                   updated = this.changed;
                   W = this.w;
                   H = this.h;
                   this.changed = false;
               }
               if (updated) {
                   if (surface != null) {
                   egl.eglMakeCurrent(dpy,
EGL10.EGL_NO_SURFACE,EGL10.EGL_NO_SURFACE, EGL10.EGL_NO_CONTEXT);
                   egl.eglDestroySurface(dpy,
 surface);
                  }
                  surface =
 egl.eglCreateWindowSurface(dpy, config, mHolder, null);
                                 egl.eglMakeCurrent(dpy, surface,
 surface, context);
            gl = (GL10) context.getGL();
            gl.glDisable(GL10.GL_DITHER);
            gl.glHint(GL10.GL_PERSPECTIVE_CORRECTION_HINT,
                      GL10.GL_FASTEST);
            gl.glClearColor(1, 1, 1, 1);
            gl.glEnable(GL10.GL_CULL_FACE);
            gl.glShadeModel(GL10.GL_SMOOTH);
            gl.glEnable(GL10.GL_DEPTH_TEST);
            gl.glViewport(0, 0, W, H);
            float ratio = (float) W / H;
            gl.glMatrixMode(GL10.GL_PROJECTION);
           gl.glLoadIdentity();
            gl.glFrustumf(-ratio, ratio, -1,
 1, 1, 10);
               }
          drawFrame(gl);
          egl.eglSwapBuffers(dpy, surface);
         if (egl.eglGetError() ==
EGL11.EGL_CONTEXT_LOST) {
             Context c = getContext();
             if (c instanceof Activity) {
                 ((Activity)c).finish();
             }
```

7 Obtain reference to OpenGL ES context

8 Do drawing

```
                }
            }
             egl.eglMakeCurrent(dpy, EGL10.EGL_NO_SURFACE,
EGL10.EGL_NO_SURFACE,
                EGL10.EGL_NO_CONTEXT);
            egl.eglDestroySurface(dpy, surface);
            egl.eglDestroyContext(dpy, context);
             egl.eglTerminate(dpy);
           }
                   public void onWindowResize(int w, int h) {
                       synchronized(this) {
                              this.w = w;
                              this.h = h;
                              this.changed = true;
                       }
                   }
                   public void waitForExit() {
                       this.stop = true;
                       try {
                           join();
                       } catch (InterruptedException ex) {
                       }
                   }
                   private void drawFrame(GL10 gl) {
                       // do whatever drawing here.
                       }
        }
    }
}
```

Listing 9.9 generates an empty black screen. Everything in this listing is code you need to draw and manage any OpenGL ES visualization. First, we import all our needed classes. Then we implement an inner class, which will handle everything about managing a surface: creating it, changing it, or deleting it. We extend the class `SurfaceView` and implement the `SurfaceHolder` interface, which allows us to get information back from Android when the surface changes, such as when someone resizes it ❶. With Android, all this has to be done asynchronously; you can't manage surfaces directly.

Next, we create a thread to do the drawing ❷ and create an `init()` method that uses the `SurfaceView` class's `getHolder()` method to get access to the `SurfaceView` and add a callback to it via the `addCallBack()` method ❸. Now we can implement `surfaceCreated()`, `surfaceChanged()`, and `surfaceDestroyed()`, which are all methods of the `Callback` class and are fired on the appropriate condition of change in the `Surface`'s state.

When all the `Callback` methods are implemented, we create a thread to do the drawing ❹. Before we can draw anything, though, we need to create an OpenGL ES context ❺ and create a handler to the `Surface` ❻ so that we can use the OpenGL context's method to act on the surface via the handle ❼. Now we can finally draw something, although in the `drawFrame()` method ❽ we aren't doing anything.

If you were to run the code right now, all you'd get would be an empty window; but what we've generated so far will appear in some form or another in any OpenGL ES application you make on Android. Typically, you'll break up the code so that an `Activity` class starts the code and another class implements the custom `View`. Yet another class may implement your `SurfaceHolder` and `SurfaceHolder.Callback`, providing all the methods for detecting changes to the surface, as well as those for the drawing of your graphics in a thread. Finally, you may have another class for whatever code represents your graphics.

In the next section, we'll look at how to draw a square on the surface and how to create an animated cube.

9.3.2 Drawing a rectangle with OpenGL ES

In the next example, you'll use OpenGL ES to create a simple drawing, a rectangle, using OpenGL primitives, which in OpenGL ES are pixels and triangles. When you draw the square, you'll use a primitive called the `GL_Triangle_Strip`, which takes three vertices (the x, y, and z points in an array of vertices) and draws a triangle. The last two vertices become the first two vertices for the next triangle, with the next vertex in the array being the final point. This process repeats for as many vertices as there are in the array, and it generates something like figure 9.5, where two triangles are drawn.

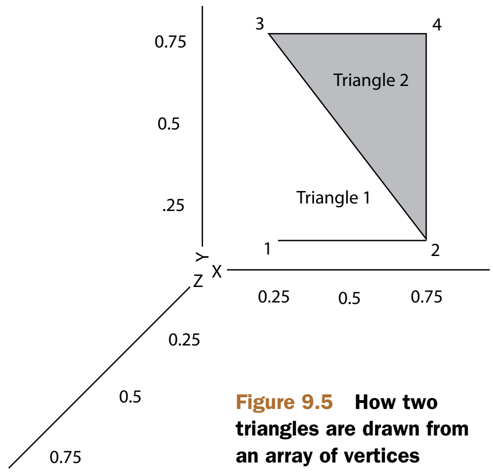


Figure 9.5 How two triangles are drawn from an array of vertices

OpenGL ES supports a small set of primitives, shown in table 9.1, that allow you to build anything using simple geometric shapes, from a rectangle to 3D models of animated characters.

Table 9.1 OpenGL ES primitives and their descriptions

Primitive flag	Description
`GL_LINE_LOOP`	Draws a continuous set of lines. After the first vertex, it draws a line between every successive vertex and the vertex before it. Then it connects the start and end vertices.
`GL_LINE_STRIP`	Draws a continuous set of lines. After the first vertex, it draws a line between every successive vertex and the vertex before it.
`GL_LINES`	Draws a line for every pair of vertices given.
`GL_POINTS`	Places a point at each vertex.
`GL_TRIANGLE_FAN`	After the first two vertices, every successive vertex uses the previous vertex and the first vertex to draw a triangle. This flag is used to draw cone-like shapes.

Table 9.1 OpenGL ES primitives and their descriptions *(continued)*

Primitive flag	Description
GL_TRIANGLE_STRIP	After the first two vertices, every successive vertex uses the previous two vertices to draw the next triangle.
GL_TRIANGLES	For every triplet of vertices, it draws a triangle with corners specified by the coordinates of the vertices.

In the next listing, we use an array of vertices to define a square to paint on our surface. To use the code, insert it directly into the code for listing 9.9, immediately below the commented line `// do whatever drawing here`.

Listing 9.10 OpenGLSquare.java

```
gl.glClear(GL10.GL_COLOR_BUFFER_BIT |
    GL10.GL_DEPTH_BUFFER_BIT);
float[] square = new float[] {
    0.25f, 0.25f, 0.0f,
    0.75f, 0.25f, 0.0f,
    0.25f, 0.75f, 0.0f,
    0.75f, 0.75f, 0.0f };
FloatBuffer squareBuff;
ByteBuffer bb =
ByteBuffer.allocateDirect(square.length*4);
    bb.order(ByteOrder.nativeOrder());
    squareBuff = bb.asFloatBuffer();
    squareBuff.put(square);
    squareBuff.position(0);
    gl.glMatrixMode(GL10.GL_PROJECTION);
    gl.glLoadIdentity();
    GLU.gluOrtho2D(gl, 0.0f,1.2f,0.0f,1.0f);
    gl.glVertexPointer(3, GL10.GL_FLOAT, 0, squareBuff);
    gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
    gl.glClear(GL10.GL_COLOR_BUFFER_BIT);
    gl.glColor4f(0,1,1,1);
    gl.glDrawArrays(GL10.GL_TRIANGLE_STRIP, 0, 4);
```

1 Create float buffer to hold square

2 Set up 2D orthographic viewing region

3 Set current vertices for drawing

This code is dense with OpenGL commands. The first thing we do is clear the screen using `glClear`, which you want to do before every drawing. Then we build the array to represent the set of vertices that make up our square. As we explained, we use the OpenGL primitive `GL_TRIANGLE_STRIP` to create the rectangle shown in figure 9.5, where the first set of three vertices (points 1, 2, and 3) represent the first triangle. The last vertex represents the third vertex (point 4) in the second triangle, which reuses vertices 2 and 3 from the first triangle as its first two to make the triangle described by points 2, 3, and 4. To put it more succinctly, Open GL ES takes one triangle and flips it over on its third side (in this case, the hypotenuse). We then create a buffer to hold that same square data 1. We also tell the system that we'll be using a `GL_PROJECTION` for our matrix mode, which is a type of matrix transformation that's applied to every point in the matrix stack.

The next things we do are more related to setup. We load the identity matrix and then use the `gluOrtho2D(GL10 gl, float left, float right, float bottom, float top)` command to set the clipping planes that are mapped to the lower-left and upper-right corners of the window ❷.

Now we're ready to start drawing the image. First, we use the `glVertexPointer(int size, int type, int stride, pointer to array)` method, which indicates the location of vertices for the triangle strip. The method has four attributes: `size`, `type`, `stride`, and `pointer`. The `size` attribute specifies the number of coordinates per vertex (for example, a 2D shape might ignore the z axis and use only two coordinates per vertex), `type` defines the data type to be used (`GL_BYTE`, `GL_SHORT`, `GL_FLOAT`, and so on) ❸, `stride` specifies the offset between consecutive vertices (how many unused values exist between the end of the current vertex and the beginning of the next), and `pointer` is a reference to the array. Although most drawing in OpenGL ES is performed by using various forms of arrays such as the vertex array, they're all disabled by default to save system resources. To enable them, we use the OpenGL command `glEnableClientState(array type)`, which accepts an array type; in this case, the type is `GL_VERTEX_ARRAY`.

Finally, we use the `glDrawArrays` function to render our arrays into the OpenGL primitives and create our simple drawing. The `glDrawArrays(mode, first, count)` function has three attributes: `mode` indicates which primitive to render, such as `GL_TRIANGLE_STRIP`; `first` is the starting index into the array, which we set to `0` because we want it to render all the vertices in the array; and `count` specifies the number of indices to be rendered, which in this case is 4.

If you run the code, you should see a simple blue rectangle on a white surface, as shown in figure 9.6. It isn't particularly exciting, but you'll need most of the code you used for this example for any OpenGL project.

There you have it—your first graphic in OpenGL ES. Next, we're going to do something way more interesting. In the next example, you'll create a 3D cube with different colors on each side and then rotate it in space.

9.3.3 *Three-dimensional shapes and surfaces with OpenGL ES*

In this section, you'll use much of the code from the previous example, but you'll extend it to create a 3D cube that rotates. We'll examine how to introduce perspective to your graphics to give the illusion of depth.

Depth works in OpenGL by using a *depth buffer*, which contains a depth value for every pixel, in the range 0 to 1. The value represents the perceived distance between objects and your viewpoint; when two objects' depth values are compared, the value closer to 0 will appear in front on the screen. To use depth in your program, you need to first enable the depth buffer by passing `GL_DEPTH_TEST` to the `glEnable()` method. Next, you use `glDepthFunc()` to define how values are compared. For this example, you'll use `GL_LEQUAL`, defined in table 9.2, which tells the system to show objects with a lower depth value in front of other objects.

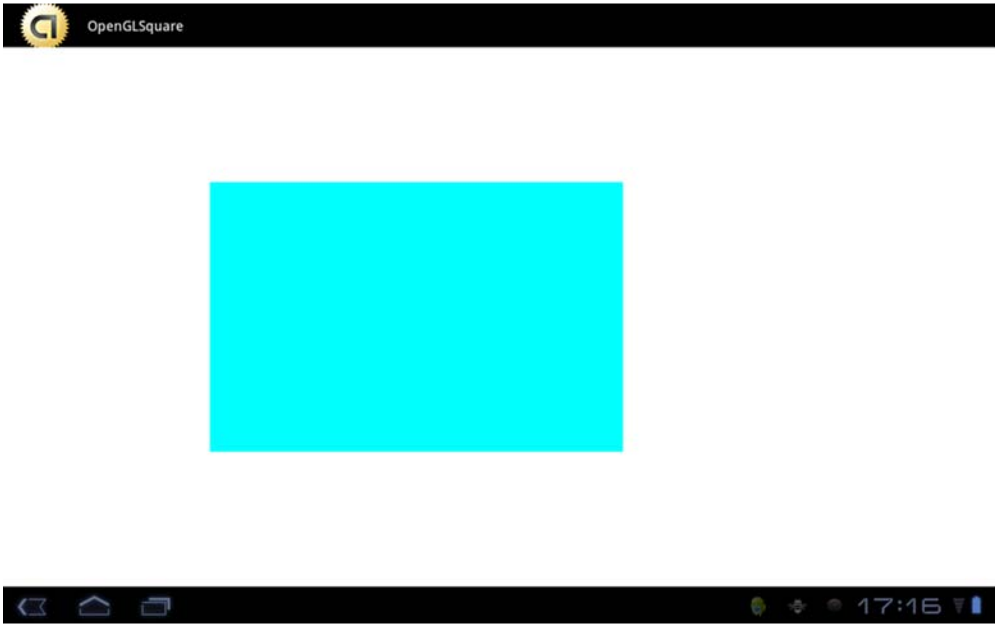


Figure 9.6 A rectangle drawn on the surface using OpenGL ES

When you draw a primitive, the depth test occurs. If the value passes the test, the incoming color value replaces the current one.

The default value is GL_LESS. You want the value to pass the test if the values are equal as well. Objects with the same z value will display, depending on the order in which they were drawn. We pass GL_LEQUAL to the function.

Table 9.2 Flags for determining how values in the depth buffer are compared

Flag	Description
GL_ALWAYS	Always passes
GL_EQUAL	Passes if the incoming depth value is equal to the stored value
GL_GEQUAL	Passes if the incoming depth value is greater than or equal to the stored value
GL_GREATER	Passes if the incoming depth value is greater than the stored value
GL_LEQUAL	Passes if the incoming depth value is less than or equal to the stored value
GL_LESS	Passes if the incoming depth value is less than the stored value
GL_NEVER	Never passes
GL_NOTEQUAL	Passes if the incoming depth value isn't equal to the stored value

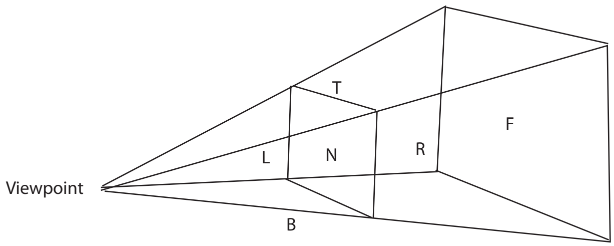


Figure 9.7 In OpenGL, a perspective is made up of a viewpoint and near (N), far (F), left (L), right (R), top (T), and bottom (B) clipping planes.

One important part of maintaining the illusion of depth is providing perspective. In OpenGL, a typical perspective is represented by a viewpoint with near and far clipping planes and top, bottom, left, and right planes, where objects that are closer to the far plane appear smaller, as in figure 9.7.

OpenGL ES provides a function called `gluPerspective(GL10 gl, float fovy, float aspect, float zNear, float zFar)` with five parameters (see table 9.3) that lets you easily create perspective.

Table 9.3 Parameters for the `gluPerspective` function

Parameter	Description
`aspect`	Aspect ratio that determines the field of view in the x direction. The aspect ratio is the ratio of x (width) to y (height).
`fovy`	Field of view angle in the y direction, in degrees.
`gl`	GL10 interface.
`zFar`	Distance from the viewer to the far clipping plane. This value is always positive.
`zNear`	Distance from the viewer to the near clipping plane. This value is always positive.

To demonstrate depth and perspective, you're going to create a project called OpenGLCube. Copy and paste the code from listing 9.11 into `OpenGLCubeActivity`.

Now add two new variables to your code, shown in the following listing, right at the beginning of the `DrawSurfaceView` inner class.

Listing 9.11 OpenGLCubeActivity.java

```
class DrawingSurfaceView extends SurfaceView implements
SurfaceHolder.Callback  {
    public SurfaceHolder mHolder;
    float xrot = 0.0f;
    float yrot = 0.0f;
```

We'll use the `xrot` and `yrot` variables later in the code to govern the rotation of the cube.

Next, just before the method, add a new method called `makeFloatBuffer()`, as in the following listing.

Listing 9.12 OpenGLCubeActivity.java

```
protected FloatBuffer makeFloatBuffer(float[] arr) {
            ByteBuffer bb = ByteBuffer.allocateDirect(arr.length*4);
            bb.order(ByteOrder.nativeOrder());
            FloatBuffer fb = bb.asFloatBuffer();
            fb.put(arr);
            fb.position(0);
            return fb;
}
```

This float buffer is the same as the one in listing 9.11, but we've abstracted it from the `drawFrame()` method so we can focus on the code for rendering and animating the cube.

Next, copy and paste the code from the following listing into the `drawFrame()` method. Note that you'll also need to update your `drawFrame()` call in the following way:

```
drawFrame(gl, w, h);
```

Listing 9.13 OpenGLCubeActivity.java

```
        private void drawFrame(GL10 gl, int w1, int h1) {
            float mycube[] = {
                // FRONT
                -0.5f, -0.5f,  0.5f,
                0.5f, -0.5f,  0.5f,
                -0.5f,  0.5f,  0.5f,
                0.5f,  0.5f,  0.5f,
                // BACK
                -0.5f, -0.5f, -0.5f,
                -0.5f,  0.5f, -0.5f,
                0.5f, -0.5f, -0.5f,
                 0.5f,  0.5f, -0.5f,
                // LEFT
                -0.5f, -0.5f,  0.5f,
                -0.5f,  0.5f,  0.5f,
                -0.5f, -0.5f, -0.5f,
                -0.5f,  0.5f, -0.5f,
                // RIGHT
                0.5f, -0.5f, -0.5f,
                0.5f,  0.5f, -0.5f,
                0.5f, -0.5f,  0.5f,
                0.5f,  0.5f,  0.5f,
                // TOP
                -0.5f,  0.5f,  0.5f,
                0.5f,  0.5f,  0.5f,
                -0.5f,  0.5f, -0.5f,
                0.5f,  0.5f, -0.5f,
                // BOTTOM
                -0.5f, -0.5f,  0.5f,
                -0.5f, -0.5f, -0.5f,
                0.5f, -0.5f,  0.5f,
                0.5f, -0.5f, -0.5f,
            };
            FloatBuffer cubeBuff;
            cubeBuff = makeFloatBuffer(mycube);
```

1 **Create float buffer for vertices**

```
            gl.glEnable(GL10.GL_DEPTH_TEST);
            gl.glEnable(GL10.GL_CULL_FACE);
            gl.glDepthFunc(GL10.GL_LEQUAL);
            gl.glClearDepthf(1.0f);
            gl.glClear(GL10.GL_COLOR_BUFFER_BIT |
GL10.GL_DEPTH_BUFFER_BIT);
            gl.glMatrixMode(GL10.GL_PROJECTION);
            gl.glLoadIdentity();
            gl.glViewport(0,0,w,h);
            GLU.gluPerspective(gl, 45.0f,
((float)w)/h, 1f, 100f);
            gl.glMatrixMode(GL10.GL_MODELVIEW);
            gl.glLoadIdentity();
            GLU.gluLookAt(gl, 0, 0, 3, 0, 0, 0, 0, 1, 0);
            gl.glShadeModel(GL10.GL_SMOOTH);
            gl.glVertexPointer(3, GL10.GL_FLOAT, 0, cubeBuff);
            gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
            gl.glRotatef(xrot, 1, 0, 0);
            gl.glRotatef(yrot, 0, 1, 0);
            gl.glColor4f(1.0f, 0, 0, 1.0f);
            gl.glDrawArrays(GL10.GL_TRIANGLE_STRIP, 0, 4);
            gl.glDrawArrays(GL10.GL_TRIANGLE_STRIP, 4, 4);
            gl.glColor4f(0, 1.0f, 0, 1.0f);
            gl.glDrawArrays(GL10.GL_TRIANGLE_STRIP, 8, 4);
            gl.glDrawArrays(GL10.GL_TRIANGLE_STRIP, 12, 4);
            gl.glColor4f(0, 0, 1.0f, 1.0f);
            gl.glDrawArrays(GL10.GL_TRIANGLE_STRIP, 16, 4);
            gl.glDrawArrays(GL10.GL_TRIANGLE_STRIP, 20, 4);
            xrot += 1.0f;
            yrot += 0.5f;
```

2 Enable depth test

3 Define perspective

4 Draw six sides in three colors

Increment x and y rotations

This listing doesn't contain much new code. First, we describe the vertices for a cube, which is built the same way as the rectangle in listing 9.10 (using triangles). Next, we set up the float buffer for our vertices 1 and enable the depth function 2 and perspective function 3 to provide a sense of depth. Note that with `gluPerspective` we passed `45.0f` (45 degrees) to give a more natural viewpoint.

Next, we use the `GLU.gluLookAt(GL10 gl, float eyeX, float eyeY, float eyeZ, float centerX, float centerY, float centerZ, float upX, float upY, float upZ)` function to move the position of the `View` without having to modify the projection matrix directly. When we've established the `View` position, we turn on smooth shading for the model and rotate the cube around the x and y axes. Then we draw the cube sides and increment the rotation so that on the next iteration of `draw()`, the cube is drawn at a slightly different angle 4. If you run the code, you should see a rotating 3D cube like the one shown in figure 9.8.

> **NOTE** You can try experimenting with the `fovy` value to see how changing the angle affects the display of the cube.

You've done a lot in this section, starting with creating an OpenGL ES context in which you can develop your OpenGL ES applications. Next, you learned how to build shapes using OpenGL ES by "triangulation" (creating multiple triangles). Then, you

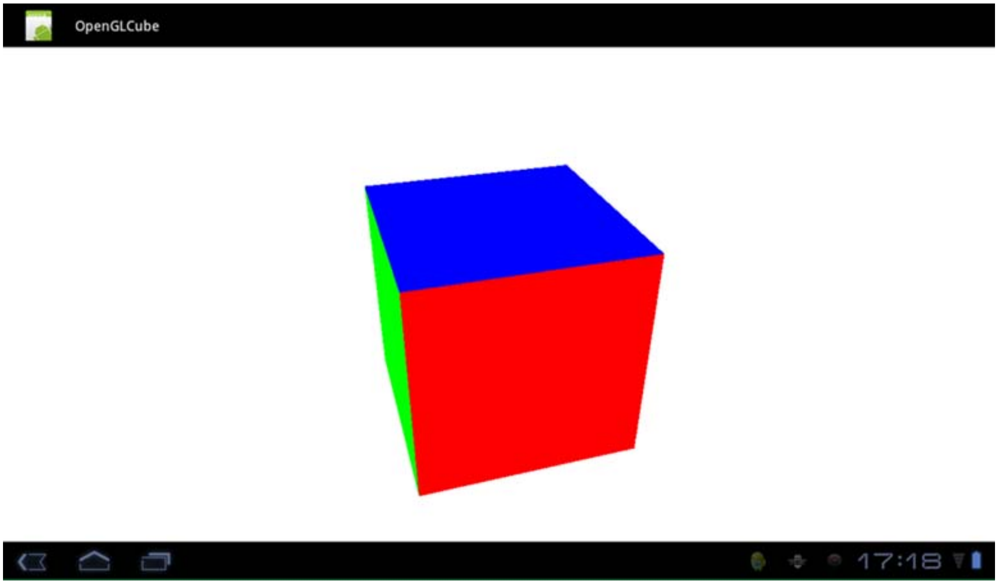


Figure 9.8 A 3D cube rotating in space

learned how to realize this in three dimensions while incorporating it into a simple example. You accomplished much of this without diving deep into OpenGL ES, which is definitely nontrivial, but the good news is that if you're serious about doing 3D graphics on Android, it's definitely possible.

With the addition of RenderScript, introduced in the next section of this chapter, developers can write code that is designed to use native code on specific hardware, allowing for much better performance of applications that are heavily dependent on processing power (such as Open GL applications). Because Android provides excellent support for OpenGL ES, you can find plenty of tutorials and references on the internet or at your local bookstore.

Now, let's look at how to use RenderScript to develop complex, rich, and high-performance graphical application that let you take advantage of the latest mobile hardware platforms that run multicore processors with dedicated graphics accelerators.

9.4 Introducing RenderScript for Android

RenderScript is a new API in Android that is focused on helping developers who need extremely high performance for graphics and computationally intensive operations. RenderScript isn't completely new to Android 3.0+; it's been part of earlier versions in 2.0 but not publicly available. As of Android 3, RenderScript has come to the fore as the tool of choice for graphically intensive games and applications such as live wallpapers, the new video carousel, and Google's e-book reader on the Xoom. In this section, we'll look at how RenderScript fits into the Android architecture, how to build a RenderScript application, and when and where to use RenderScript.

RenderScript in many ways is a new paradigm for the Android platform. Although Android uses Java syntax and a virtual machine for developing applications, RenderScript is based on C99, a modern dialect of the C language. Furthermore, RenderScript is compiled down to native code on each device at runtime but is controlled by higher-level APIs running in the Android VM. This allows Android via RenderScript to provide developers a way to develop optimized high-performance code that is cross platform. This may seem extremely attractive, and many developers may be keen to write most of their applications in RenderScript, but RenderScript doesn't replace or subsume development of Android apps in Java. There are both pros and cons to working with RenderScript.

9.4.1 RenderScript advantages and disadvantages

As already discussed, the first advantage of using RenderScript is that it's a lower-level language offering higher performance. Second, it allows Android apps to more easily use multicore CPUs as well as graphical processing units (GPUs). RenderScript, by design, at runtime selects the best-performance approach to running its code. This includes running the code across multiple CPUs; running some simpler tasks on GPUs; or, in some cases where no special hardware is present, running on just one CPU.

RenderScript offers fantastic performance and cross-platform compatibility without the need to target specific devices or create your own complex architectures for cross-platform compatibility. RenderScript is best for two types of applications and only has APIs to support those two types of applications: graphical applications and computationally intensive applications. Many applications that use Android's implementation of OpenGL are good candidates to target for RenderScript.

The first major drawback of RenderScript is that it uses C99. Although there is nothing wrong with C99, it breaks the Java style paradigm that most Android developers are comfortable with. To be truly comfortable developing RenderScript applications, you should also be comfortable with C, a lower-level language when compared to Java.

Second, and perhaps most important, RenderScript applications are inherently more complex and difficult to develop than regular Android applications. In part this is because you're developing in two different languages, Java and C; but in addition, RenderScript by its nature is very hard to debug—at times frustratingly so, unless you have a strong understanding of both your application and the hardware it's running on. For example, if you have a multicore platform with a GPU, your code may be run on either the CPUs or the GPU, reducing your ability to spot issues. Also be aware that most RenderScript applications won't run in the emulator, forcing you to debug on hardware as well.

Finally, you'll find that you have a lot more bugs, because RenderScript is in C, the current Android Development Tools (ADT) application for Eclipses doesn't support the various extensions for it, and RenderScript applications tend to be more complex than regular Android applications. But you shouldn't avoid developing in

RenderScript, nor should you overuse it as opposed to the standard Android APIs and Java syntax. Rather, you should look to use RenderScript in applications that are graphically intensive or computationally intensive.

Let's try building a RenderScript application.

9.4.2 Building a RenderScript application

Building a RenderScript application is a bit more complicated than developing a normal Android application. You lay out your application in a similar manner, but keep in mind that you'll be also developing RenderScript files, with the .rs file extension, alongside your .java files. Your normal .java application files then call the RenderScript code as needed; when you build your project, you'll see the .rs files built into bytecode with the same name as the RenderScript file but with the .bc extension under the raw folder. For example, if you had a RenderScript file called Helloworld.rs under src, you'd see a Helloworld.bc file when your application was built.

> **NOTE** We won't be covering the C or C99 language; we assume you know C. If you don't know C, you'll need to reference another resource such as Manning's *C# in Depth*, 2nd edition, by John Skeet.

For your RenderScript application, you're going to use the ADT's built-in Android project wizard to create a RenderScript project from built-in sample applications. To do so, first create a new project using the ADT, but instead of selecting Create New Project in Workspace, select Create Project from Existing Sample, as shown in figure 9.9. Make sure you've selected API level of 11 or Android 3.0, and select the sample RenderScript > Fountain from the Samples drop-down list. Click OK.

Eclipse now builds the RenderScript application. Expand the application in the Eclipse Package Explorer, as shown in figure 9.10. There are several things to note here before we go over each file. First, note the RenderScript file with the extension .rs. This is a file written in C. This file does all the real graphics work, and the other .java files provide the higher-level calls to APIs to set up a `View`, manage inputs, and the like. This file is compiled when the project is built into bytecode, which you can see when you expand the raw directory.

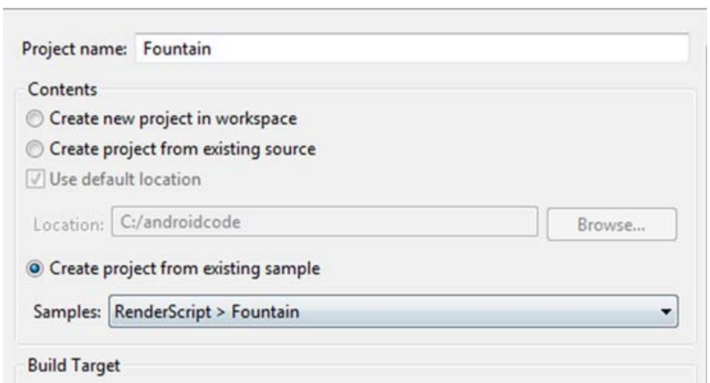


Figure 9.9 Using the ADT to build a sample RenderScript application

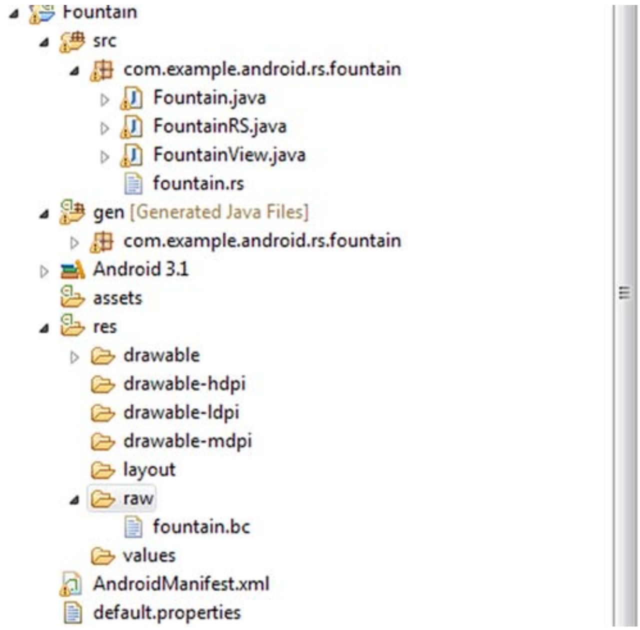


Figure 9.10 The Fountain project in the Eclipse Package Explorer showing a typical RenderScript application structure

Now that we've touched on the file layout, let's look at the source code. The first file, Fountain.java, is trivial: it's the basic Android `Activity` class. As you can see in the following listing, it has an `onCreate()` method that sets the `contentView` to an instance of the `FountainView` class.

Listing 9.14 Basic Android `Activity` class

```
public class Fountain extends Activity {

    private static final String LOG_TAG = "libRS_jni";
    private static final boolean DEBUG  = false;
    private static final boolean LOG_ENABLED = DEBUG ? Config.LOGD :
➥Config.LOGV;
    private FountainView mView;
    public void onCreate(Bundle icicle) {
        super.onCreate(icicle);
        mView = new FountainView(this);
        setContentView(mView);
    }

    protected void onResume() {
        Log.e("rs", "onResume");
        super.onResume();
        mView.resume();
    }

    protected void onPause() {
        Log.e("rs", "onPause");
        super.onPause();
        mView.pause();
    }
```

```
    static void log(String message) {
        if (LOG_ENABLED) {
            Log.v(LOG_TAG, message);
        }
    }
}
```

The FountainView.java file introduces a new type of Android `View`, the `RSSurfaceView`, as you can see in the next listing. This class represents the `SurfaceView` on which your RenderScript code will draw its graphics.

Listing 9.15 `RSSurfaceView`

```
public class FountainView extends RSSurfaceView {

    public FountainView(Context context) {
        super(context);
    }

    private RenderScriptGL mRS;
    private FountainRS mRender;

    public void surfaceChanged(SurfaceHolder holder, int format, int w,
➥int h) {
        super.surfaceChanged(holder, format, w, h);
        if (mRS == null) {
            RenderScriptGL.SurfaceConfig sc = new
➥RenderScriptGL.SurfaceConfig();
            mRS = createRenderScriptGL(sc);
            mRS.setSurface(holder, w, h);
            mRender = new FountainRS();
            mRender.init(mRS, getResources(), w, h);
        }
    }

    protected void onDetachedFromWindow() {
        if (mRS != null) {
            mRS = null;
            destroyRenderScriptGL();
        }
    }

    public boolean onTouchEvent(MotionEvent ev)
    {
        int act = ev.getActionMasked();
        if (act == ev.ACTION_UP) {
            mRender.newTouchPosition(0, 0, 0, ev.getPointerId(0));
            return false;
        } else if (act == MotionEvent.ACTION_POINTER_UP) {
            int pointerIndex = ev.getActionIndex();
            int pointerId = ev.getPointerId(pointerIndex);
            mRender.newTouchPosition(0, 0, 0, pointerId);
        }
        int count = ev.getHistorySize();
        int pcount = ev.getPointerCount();

        for (int p=0; p < pcount; p++) {
```

1 Create new RenderScript

2 Create the FountainRS class

3 Handle touch events

```
                int id = ev.getPointerId(p);
                mRender.newTouchPosition(ev.getX(p),
                                         ev.getY(p),
                                         ev.getPressure(p),
                                         id);

                for (int i=0; i < count; i++) {
                    mRender.newTouchPosition(ev.getHistoricalX(p, i),
                                             ev.getHistoricalY(p, i),
                                             ev.getHistoricalPressure(p, i),
                                             id);
                }
            }
            return true;
        }
}
```

If you look at the listing, you'll notice in the `surfacedChanged()` method a new RenderScript class as well as a `FountainRS` class ❶. The code

```
RenderScriptGL.SurfaceConfig sc = new RenderScriptGL.SurfaceConfig();
        mRS = createRenderScriptGL(sc);
```

is important in that it not only creates a `RenderScriptGL` object that contains the surface our graphics go into, but the `SurfaceConfig` class allows us to set all the major properties for the drawing surface (such as depth). The `FountainRS` class is important in that it acts as a renderer for the `FountainView` ❷ `SurfaceView` as well as controls the actual RenderScript. One of the other important things this `FountainView` class does is handle touch events with the `onTouchEvent()` method and pass these events to the RenderScript ❸.

The next class we'll look at is `FountainRS`, shown in the following listing.

Listing 9.16 `FountainRS` class

```
public class FountainRS {
    public static final int PART_COUNT = 50000;

    public FountainRS() {
    }
    private Resources mRes;
    private RenderScriptGL mRS;
    private ScriptC_fountain mScript;
    public void init(RenderScriptGL rs, Resources res,
➥int width, int height) {
        mRS = rs;
        mRes = res;
        ProgramFragmentFixedFunction.Builder pfb = new
     ProgramFragmentFixedFunction.Builder(rs);
        pfb.setVaryingColor(true);
        rs.bindProgramFragment(pfb.create());
        ScriptField_Point points =                          ❶ Bind ScriptC_fountain class
➥new ScriptField_Point(mRS, PART_COUNT);
        Mesh.AllocationBuilder smb = new Mesh.AllocationBuilder(mRS);
        smb.addVertexAllocation(points.getAllocation());
```

```
        smb.addIndexSetType(Mesh.Primitive.POINT);
        Mesh sm = smb.create();

        mScript = new ScriptC_fountain(mRS, mRes, R.raw.fountain);
        mScript.set_partMesh(sm);
        mScript.bind_point(points);
        mRS.bindRootScript(mScript);
    }

    boolean holdingColor[] = new boolean[10];
    public void newTouchPosition(float x, float y,
➥float pressure, int id) {
        if (id >= holdingColor.length) {
            return;
        }
        int rate = (int)(pressure * pressure * 500.f);
        if (rate > 500) {
            rate = 500;
        }
        if (rate > 0) {
            mScript.invoke_addParticles(rate, x, y, id, !holdingColor[id]);
            holdingColor[id] = true;
        } else {
            holdingColor[id] = false;
        }

    }
}
```

When developing a graphical RenderScript application, you'll have a class called `ClassNameRS` that acts as a communication channel between your RenderScript file and the rest of the Android application. (RenderScript compute projects don't have a file like this.) The `FountainRS` class interacts with the RenderScript code in fountain.rs via interfaces exposed by `ScriptC_fountain`, a class generated by the ADT when you build the project and found in the gen folder. The `ScriptC_fountain` class binds to the RenderScript bytecode so the `RenderScriptGL` context knows which RenderScript to bind to ❶. This may sound somewhat complicated, and it is, but the ADT or Android tooling manages most of this for you.

Finally, let's look at the C code in fountain.rs, shown in listing 9.17. The first thing you'll notice is how simple it is. The code draws a simple cascade of points whose center is the point touched on the screen. It's important to note that all the methods to capture the information about where the user presses are captured, handled, and passed down to this class via the higher-level .java classes already discussed, and that fountain.rs is solely focused on drawing.

Listing 9.17 C code in fountain.rs

```
#pragma version(1)
#pragma rs java_package_name(com.example.android.rs.fountain)
#pragma stateFragment(parent)
#include "rs_graphics.rsh"

static int newPart = 0;
```

Required pragma directives class

```
rs_mesh partMesh;

typedef struct __attribute__((packed, aligned(4))) Point {
    float2 delta;
    float2 position;
    uchar4 color;
} Point_t;
Point_t *point;

int root() {
    float dt = min(rsGetDt(), 0.1f);
    rsgClearColor(0.f, 0.f, 0.f, 1.f);
    const float height = rsgGetHeight();
    const int size = rsAllocationGetDimX(rsGetAllocation(point));
    float dy2 = dt * (10.f);
    Point_t *p = point;
    for (int ct=0; ct < size; ct++) {
        p->delta.y += dy2;
        p->position += p->delta;
        if ((p->position.y > height) && (p->delta.y > 0)) {
            p->delta.y *= -0.3f;
        }
        p++;
    }

    rsgDrawMesh(partMesh);
    return 1;
}

static float4 partColor[10];
void addParticles(int rate, float x, float y, int index, bool newColor)
{
    if (newColor) {
        partColor[index].x = rsRand(0.5f, 1.0f);
        partColor[index].y = rsRand(1.0f);
        partColor[index].z = rsRand(1.0f);
    }
    float rMax = ((float)rate) * 0.02f;
    int size = rsAllocationGetDimX(rsGetAllocation(point));
    uchar4 c = rsPackColorTo8888(partColor[index]);

    Point_t * np = &point[newPart];
    float2 p = {x, y};
    while (rate--) {
        float angle = rsRand(3.14f * 2.f);
        float len = rsRand(rMax);
        np->delta.x = len * sin(angle);
        np->delta.y = len * cos(angle);
        np->position = p;
        np->color = c;
        newPart++;
        np++;
        if (newPart >= size) {
            newPart = 0;
            np = &point[newPart];
        }
    }
}
```

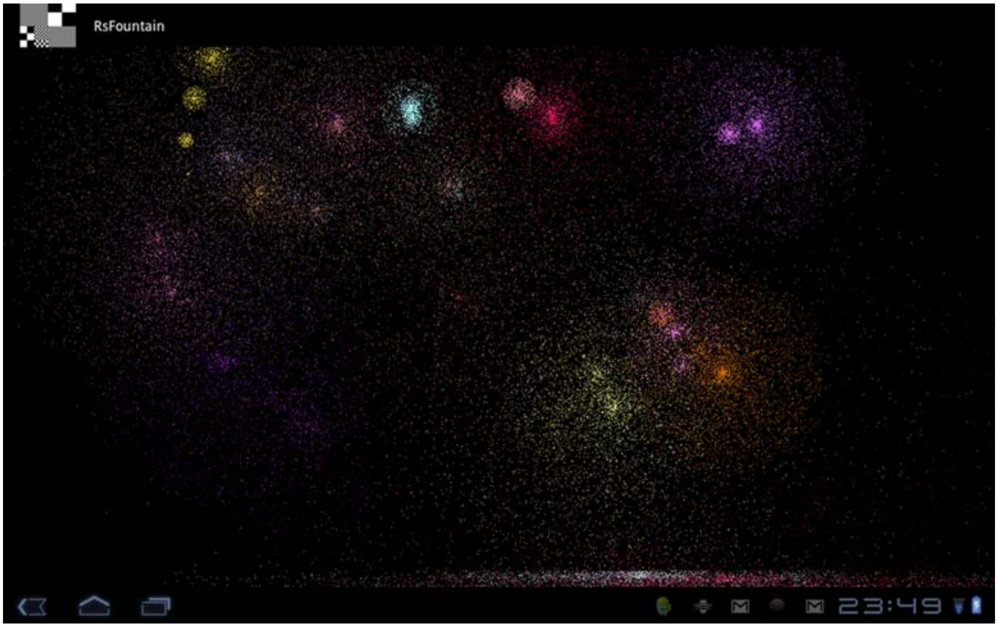


Figure 9.11 Example of the Fountain project running on the Xoom

The first thing to note is the inclusion of two pragmas that must be part of any RenderScript file, which provide the version and package name. Also note the use of two functions familiar to C developers, `init()` and `root()`. The `init()` function provides a mechanism for setting up variables or constants before anything else is executed in the class. The `root()` method is of course the main root function of the class; for graphics applications, RenderScript will expect to render the frame to be displayed in this method. Other than that, the C code is relatively straightforward.

If you run this application and then touch the screen, you should see a burst of color and cascading dots that fall to the bottom of the screen as shown in figure 9.11. Although you could have done the same thing with Android's 2-D API, and it would have been much easier to code, the RenderScript application is extremely fast with no discernable lag on a Motorola Xoom.

We can't go into RenderScript in depth in this book—it warrants its own chapter—but we've touched on the main points. You now know the basics of how to build your own RenderScript graphical applications.

9.5 Summary

In this chapter, we've lightly touched on a number of topics related to Android's powerful graphics features. First, we looked at how both Java and XML can be used with the Android Graphics API to describe simple shapes. Next, we examined how to use Android's frame-by-frame XML to create an animation. You also learned how to use more standard pixel manipulation to provide the illusion of movement through Java and the Graphics API. Finally, we delved into Android's support of OpenGL ES. We looked at how to create an OpenGL ES context, and then we built a shape in that context as well as a 3D animated cube. Finally, we took a high-level look at a RenderScript application and discussed how the RenderScript system works inside Android.

Graphics and visualizations are large and complex topics that can easily fill a book. But because Android uses open and well-defined standards and supports an excellent API for graphics, it should be easy for you to use Android's documentation, API, and other resources, such as Manning's *Java 3D Programming* by Daniel Selman, to develop anything from a new drawing program to complex games.

In the next chapter, we'll move from graphics to working with multimedia. We'll explore working with audio and video to lay the groundwork for making rich multimedia applications.

MOBILE TECHNOLOGY

Android IN ACTION THIRD EDITION

Ableson • Sen • King • Ortiz

Free ebook
SEE INSERT

When it comes to mobile apps, Android can do almost anything—and with this book, so can you! Android, Google's popular mobile operating system and SDK for tablets and smart phones, is the broadest mobile platform available. It is Java-based, HTML5-aware, and loaded with the features today's mobile users demand.

Android in Action, Third Edition takes you far beyond "Hello Android." You'll master the SDK, build WebKit apps using HTML 5, and even learn to extend or replace Android's built-in features. You'll find interesting examples on every page as you explore cross-platform graphics with RenderScript, the updated notification system, and the Native Development Kit. This book also introduces important tablet concepts like drag and drop, fragments, and the Action Bar, all new in Android 3.

What's Inside

- Covers Android 3.x
- SDK and WebKit development from the ground up
- Driving a robot with Bluetooth and sensors
- Image processing with Native C code

This book is written for hobbyists and developers. A background in Java is helpful—no prior experience with Android is assumed.

Frank Ableson and **Robi Sen** are entrepreneurs focused on mobile and web products, and on novel wireless technologies, respectively. **Chris King** is a senior mobile engineer and **C. Enrique Ortiz** a mobile technologist, developer, and author.

"Gold standard of Android training books."
—Gabor Paller, Ericsson

"Still the best single book for both beginners and experts."
—Matthew Johnson
Sabaki Engineering

"Fully covers most Android tablet functionalities."
—Loïc Simon, SII

For access to the book's forum and a free ebook for owners of this book, go to manning.com/AndroidinActionThirdEdition

ISBN 13: 978-1-61729-050-3
ISBN 10: 1-61729-050-5
54999
9 781617 290503

MANNING $49.99 / Can $52.99 [INCLUDING eBOOK]