PART 3

Android: Let's get moving

LINUX FORMAT On the DVD
» Code and all the previous tutorials

Our expert

Juliet Kemp found moving a bubble around a screen to be a surprisingly satisfying experience.

Juliet Kemp shows you how to make the most of your phone's 'mobile' nature by coding a simple spirit level app.

In the previous two parts of this tutorial, we looked at setting up a fairly basic Android program, displaying information, and getting some user input. We also covered a few parts of the fairly extensive Android API.

This time, we're going to start looking at some of the really neat aspects of writing code for a portable device.

Unlike a desktop, or even a laptop PC, phones are very mobile. You can shake them around and turn them in different directions – and, when you move the phone around in these ways, it means that you can get your code to work differently.

The Android API includes a bunch of different sensors which gather this information, and which you can use to inform your code, and this tutorial is going to look at the gravity sensor – and its relation to the accelerometer. But first, a closer look at Android graphics.

Graphics

In previous tutorials we used View objects to display and arrange information on the screen. Views are the basic building blocks of the user interface, handling drawing and events within a rectangular part of the screen. You can set up trees and hierarchies of Views to create more complicated layouts, and you can show drawings and animations by placing them into a View object, but Views are only really suitable for static content.

If you're writing an app with graphics which need to respond regularly to user interaction – for example if your screen will be regularly redrawing – a better option is to use a Canvas. A Canvas provides an interface between you and the bitmap which will actually be drawn onto the area controlled by it. In this tutorial we'll create a spirit level app, which redraws itself when the user tilts the device, so a Canvas is the best option.

Two basic ways to use a Canvas:

1. Create your own custom View component, and handle the redrawing via this. This option is fine if the app doesn't require a lot of processor power, and doesn't need to redraw itself very quickly.
2. Use a separate thread and a SurfaceView. This can redraw the screen as fast as the thread will go, so this is good for apps which need a fast response.

The spirit level app doesn't need to redraw itself at high speed, so in this case, we'll use the first option, and create a custom View component.

To set up the new project, type:

```
android create project --target android-10
--name spiritlevel \\
--path ~/android/spiritlevel --activity
SpiritLevelActivity \\
--package com.example.spiritlevel
```

(See previous tutorials, or the Android online docs, for more explanation.) Note that this time, we're developing against Android 2.3.3 (API level 10), as the gravity sensor was only introduced in API level 9.

The first step is to set up our custom View, which will draw the basic spirit level outline. Not all the code will be included in the text so see the coverdisc for the full listing. The code for this first pass of the graphics setup, which we'll improve later in the tutorial, is saved as **SpiritLevelDrawableViewOne.java**, so you'll need to rename it **SpiritLevelDrawableView.java** for it to compile properly.

The code for the outer box of the spirit level looks like this:

```
public class SpiritLevelDrawableView extends
View {
  private LayerDrawable layer;
  public SpiritLevelDrawableView(Context
context) {
    super(context);
    int outerX = 80;
    int outerY = 50;
    int outerWidth = 150;
    int outerHeight = 300;
    ShapeDrawable outer = new
ShapeDrawable();
    outer.getPaint().setColor(0xFF9933FF);
    outer.setBounds(outerX, outerY, outerX +
outerWidth,
        outerY + outerHeight);
            layer = new
LayerDrawable(new Drawable[] {outer});
  }
  protected void onDraw(Canvas canvas) {
    layer.draw(canvas);
  }
}
```

A **Drawable** object is a catch-all class for 'something that can be drawn'; it's extended to classes like **ShapeDrawable** and **LayerDrawable** to define more specific things. You can also extend it yourself to create your own custom drawable objects.

The new **ShapeDrawable** variable, **outer** (the outside of the spirit level) will be constructed using the default of a **RectShape**, since the constructor has no other argument.

Next, we set the colour, using an RGB value, then the shape's boundaries. See the boxout for more on colour – this one is a pleasingly garish purple, but you might have something more soothing in mind. **ShapeDrawable** will draw the Shape it has constructed (here, the default **RectShape**) to the boundaries provided by the **setBounds** method.

This means giving the x and y coordinates of the shape's top left corner, and then the x and y coordinates of its bottom right corner. Basically, you specify a rectangle on the screen, and the **ShapeDrawable** is drawn within that.

On the Android screen, 2D coordinates start from the top left corner of the screen and are measured in pixels. This is the case whether the display is in portrait or landscape, but which physical corner of the screen the 'top left' one is, will of course change depending on which way you hold your phone.

For this app, we'll want to lock the orientation so that turning the phone around doesn't move the spirit level. This also means that we don't need to worry about coding our layout to behave well when the screen orientation changes.

To do this, edit **AndroidManifest.xml** to add a screen orientation attribute to the **activity** element:

```
<activity android:name="SpiritLevelActivity"
  android:label="@string/app_name"
  android:screenOrientation="portrait">
```

The **setBounds(Rect bounds)** method is a method on **ShapeDrawable**, inherited from **Drawable**. Once **outer** is fully defined, the next step is to create a **LayerDrawable**. If we only needed to draw the single rectangle of **outer**, this wouldn't be necessary, but as we'll also want an **inner** box (for the spirit level bubble to bounce around in) and the bubble itself, the best bet is to use a **LayerDrawable**, constructed from an array of **Drawable** objects. It's declared as a class variable because we'll revisit it later in the code. The contents of the **LayerDrawable** are drawn on top of each other in order, the first element of the array first.

Bubble building

Finally, a View needs an **onDraw** method to define what happens when it is drawn. Here, that's straightforward: just use the **draw** method of the LayerDrawable:

```
protected void onDraw(Canvas canvas) {
            layer.draw(canvas);
}
```

Check the code on the disk for the inner box (the 'liquid' of the spirit level, that the bubble floats within). For the bubble, we'll use an **OvalShape**, and define its bounds as a square, to produce a circle:

```
bubble = new ShapeDrawable(new OvalShape());
bubble.getPaint().setColor(0xFF000000);
bubble.setBounds(bubbleX, bubbleY, bubbleX +
bubbleDiam,
  bubbleY + bubbleDiam);
layer = new LayerDrawable(new Drawable[] {outer, liquid,
bubble});
```

Now the custom View is set up, the main application method, **SpiritLevelActivity**, needs to use it:

```
public class SpiritLevelActivity extends Activity
{
            SpiritLevelDrawableView spiritlevelView;
  @Override
  public void onCreate(Bundle savedInstanceState)
  {
    super.onCreate(savedInstanceState);
                    spiritlevelView = new
SpiritLevelDrawableView(this);
    setContentView(spiritlevelView);
  }
}
```

Compile your activity with **ant debug**, fire up **android** and start a test device, install the new code with ant **install -rbin/spiritlevel-debug.apk**, and give it a go. Behold! A very basic graphical spirit level. Which so far, of course, does absolutely nothing of any interest. On to the next bit of the code...

Android devices come in many different sizes and screen densities; so in general it's not a good idea to hard-code pixels like this in your graphics code. There are a couple of ways around this, including the use of **dp** (density independent pixels) units, using XML layouts, and providing alternative bitmaps for different screen densities. We'll look at some of these in later tutorials. One quick way to make things scale about right on different screens is to set **android:anyDensity** in your **AndroidManifest.xml** file:

```
<manifest ... >
  <supports-screens android:anyDensity="false" />
</manifest>
```

This means that the system scales any absolute pixel coordinates at runtime, rather than pre-scaling them, according to the reported screen density. For now, it's a quick way to minimise problems; but you shouldn't release code into the wild with hard-coded pixels.

»

Using colo(u)r

Colours in Android are defined as packed ints, with four bytes, one each for alpha (transparency), red, green and blue, each in hex. An alpha value of FF is opaque (so 00 would be entirely transparent). Opaque red would be 0xFFFF0000; opaque black 0xFF000000.

The **Color** class gives you a few constants, such as **Color.BLACK** and **Color.YELLOW**, but that's a pretty limited colour palette, so you'll probably want to make at least some use of the packed integers.

However, having bare ints like this in code isn't a great idea – it's not very maintainable. Instead, you can store your color definitions in a resources file, such as **res/values/colors.xml** (the filename doesn't matter, although it does need to be in the **res/values/** directory):

```
 <?xml version="1.0"
encoding="utf-8"?>
 <resources>
    <color
name="brightpurple">#ff9933ff</color>
    <color
name="brightyellow">#ffffff00</color>
 </resources>
```

You can then refer to these resources in your code like this:

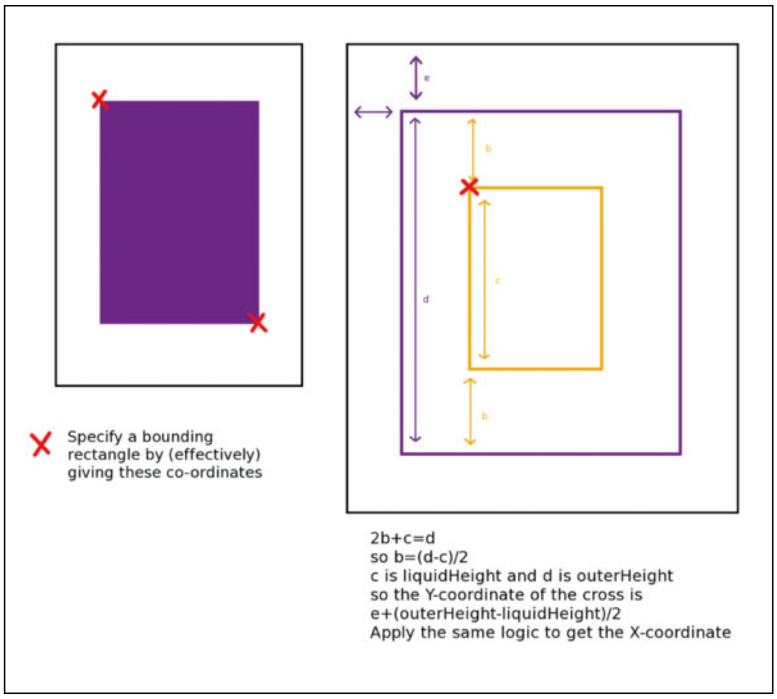
```
 Resources res = getResources();
 outer.getPaint().setColor(res.getColor(R.
color.brightpurple));
```

If you're using one of the colours that does have a constant from **Color** assigned to it, it's good practice (and saves code lines) to use that instead of self-defining your colour:

```
 bubble.getPaint().setColor(Color.
BLACK);
```

Quick tip

If **adb** says 'device offline' the first time you try to install your test code, just run it again.



Specify a bounding rectangle by (effectively) giving these co-ordinates

2b+c=d
so b=(d-c)/2
c is liquidHeight and d is outerHeight
so the Y-coordinate of the cross is
e+(outerHeight-liquidHeight)/2
Apply the same logic to get the X-coordinate

› **Calculating your graphics values.**

» Android has a bunch of different hardware sensors: accelerometer, gravity, gyroscope, light, linear acceleration, magnetic field, pressure, proximity, rotation vector and temperature. Each can be represented by the **Sensor** class, and accessed by the **SensorManager** class. Sensor events are represented by the **SensorEvent** class, which holds data (including timestamp and accuracy) about each sensor event.

The **SensorEvent** class uses a different set of coordinates from the 2D graphics engine. The origin is in the middle of the screen; the x-axis is horizontal (across the screen) and points right; the y-axis is vertical (up and down the screen) and points up; and the z-axis runs through the middle of the phone and points outwards from the front of the screen.

To see how the **Sensor**, **Management**, and **Event** classes work together, let's set up the gravity sensor:

```
public class SpiritLevelActivity extends Activity implements
SensorEventListener {
            private SpiritLevelDrawableView spiritlevelView;
            private SensorManager manager;
            private Sensor gravity;
   @Override
   public void onCreate(Bundle savedInstanceState)
   {
      super.onCreate(savedInstanceState);
            spiritlevelView = new
SpiritLevelDrawableView(this);
                        setContentView(spiritlevelView);
                        manager = (SensorManager)
getSystemService(SENSOR_SERVICE);
      gravity = manager.getDefaultSensor(Sensor.TYPE_
GRAVITY);
   }
```

The first section of the code sets up the custom View. Then the **getSystemService** method (inherited from the **Activity** class) gets the sensor manager service, which we can then use to get a gravity sensor.

Talking sensors

You could leave the sensors on all the time, but that uses up the battery alarmingly fast. It's good practice to turn them off when the activity is paused, and pick them back up on resume:

```
    protected void onResume() {
       super.onResume();
       manager.registerListener(this, gravity,
SensorManager.SENSOR_DELAY_GAME);
    }
    protected void onPause() {
       super.onPause();
       manager.unregisterListener(this);
    }
```

Alternatives for the sensor delay rate are **SENSOR_DELAY_NORMAL** and **SENSOR_DELAY_FASTEST**, but this one should work OK for our purposes. Finally, we need to do something when the sensor data changes:

```
    public void onSensorChanged(SensorEvent event) {
spiritlevelView.invalidate();
    }
```

All this does is redraw the spirit level View. Which is great, but what we want is for that View to reflect the data coming in from the sensor, which right now it doesn't. So the next step is to grab that data and to do something useful with it.

The sensor returns an array of three values, representing a 3D vector (x, y, z) showing the direction and magnitude of gravity. First, let's ignore the x-value and look only at tilt in the y direction (tilt up and down; the x-direction is side-to-side).

If the device is lying absolutely flat, then gravity acts straight down the z-axis, meaning that the z-value is 9.81 (9.81m/s^2, the magnitude of gravity on the Earth), and the y value is zero. If, on the other hand, the device is balancing on its top edge, the gravity runs straight along the y-axis, so the y-value is 9.81 and the z-value is zero. If the device is anywhere between those two states, then y and z will be somewhere between those two sets of values.

The easiest option is to map the y-value directly onto the location of the bubble, and redraw it accordingly as the device's gravity alignment changes.

Let's put a method into **SpiritlevelDrawableView** to do that. First, we'll have to move all the bubble's dimensions outside the main method, so that we can change them from the **moveBubble()** method. This defines the bubble's default position, in the centre of the inner box. Now, create a new **moveBubble()** method:

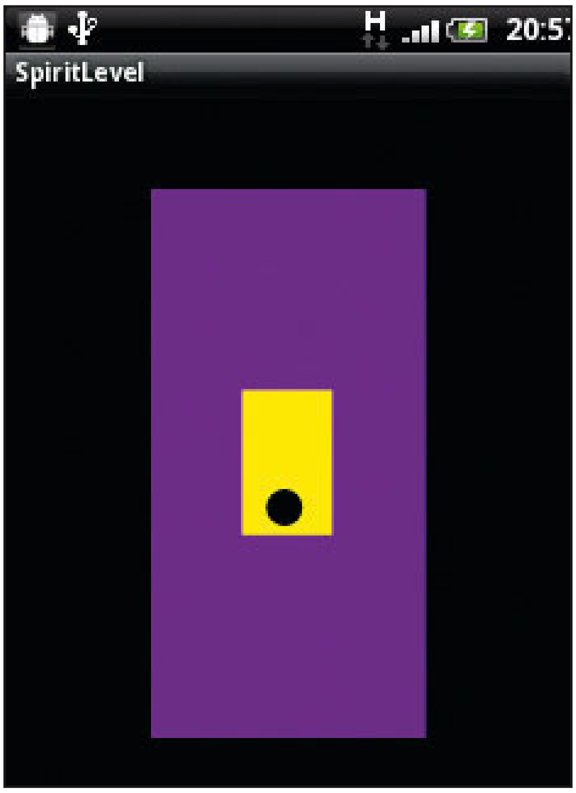
```
protected void moveBubble(float gravY) {
            bubbleY = bubbleOrigY - (int) (gravY * 5);
}
```

and redraw the bubble in the **onDraw()** method (this is called when **invalidate()** is called on the View):

```
   protected void onDraw(Canvas canvas) {
      bubble.setBounds(bubbleX, bubbleY, bubbleX +
bubbleDiam,
bubbleY + bubbleDiam);
      layer.draw(canvas);
   }
```

In **moveBubble**, the sensor value is passed in as a float, but must be cast to an int before saving it in **bubbleY**, as the **setBounds** method called in **onDraw** requires ints.

Also, because the y-axis for the graphics goes up in value as you move down the screen, to get the bubble moving in the correct direction, the gravity value must be subtracted from its origin. (Feel free to experiment with this to find out for yourself.) Finally, multiplying by five gives us more



› **Testing the spirit level.**

Quick tip

When the device is at rest, the output of the accelerometer and the gravity sensor should be identical. So if you want to test this code on a device running Android 2.2 or earlier, which doesn't have a gravity sensor, just swap **TYPE_ACCELEROMETER** for **TYPE_GRAVITY** in the code, and make sure you hold it still when testing.

movement in the bubble – as the sensor value has a min of 0 and a max of 9.81, moving the bubble by only those bare values would make for a very small movement (only 10 pixels in either direction).

Values and logging

To find out what values the sensors are returning, and to experiment with pixel values for drawing, you might want to try some logging and debugging. To add a log line in your code, use this:

```
/* this line at top of class */
private static final String TAG = "SpiritLevelActivity";
/* this line where you want your debugging */
 Log.i(TAG, "onSensorChanged() " + event.values[0] + " " +
                    event.values[1] + " " + event.values[2] + "
 " + gravY);
```

Fire up *ddms* from the command line, and run your app. In the left-hand pane of *ddms*, you'll see process info for any devices and emulators you have hooked up. Click on the device you want to debug, run your activity, and debugging info will appear in the pane at the bottom. This is obviously particularly handy if you're experiencing run-time crashes, as it'll give you a stack trace.

Now we'll get back to the **SpiritLevelActivity** class. Grab the sensor information when the sensor changes, call this method, then redraw the View:

```
public void onSensorChanged(SensorEvent event) {
  spiritlevelView.moveBubble(event.values[1]);
  spiritlevelView.invalidate();
}
```

Unfortunately, you can't use the device emulator to test this stuff, as it doesn't provide fake sensor data. You'll need to hook your device up to your laptop and test that way. See the last tutorial, or **http://developer.android.com/guide/developing/device.html** for details.

Once the y-values work, you can put in very similar code for the x-values; although in this case, you should add rather than subtract the gravity value in **moveBubble()** to get movement in the correct direction:

```
bubbleX = bubbleOrigX + (int) (gravX * 5);
```

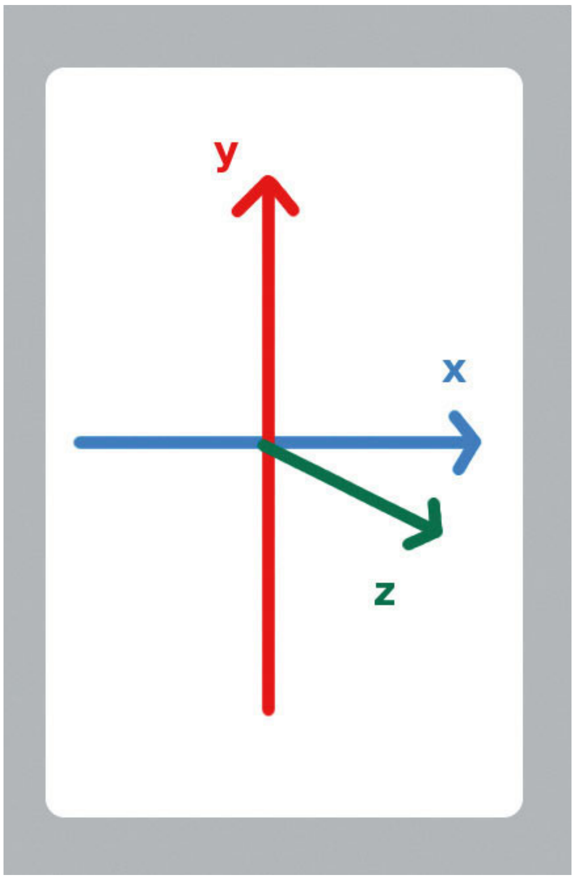
Calculating graphics

At this point, there's still quite a lot of hard-coded graphics values, and they're mostly calculated by trial and error. While we won't take all of the pixel values out, we can make it all fit together a little bit better.

The liquid box needs to have enough room for the bubble to move around in, but no more than that – when the phone is tipped all the way in one direction, the bubble should be at the very end of the box (exactly as you would see with a real spirit level).

Happily, we know exactly how much the bubble will move in each direction: the maximum possible of the raw gravity value (**gravX** or **gravY**), multiplied by five as our magnifying factor. It's bad practice to have the magnifying factor in the code, as well, so:

```
private static final double WIDTH_MAGNIFY = 1.5;
private static final double HEIGHT_MAGNIFY = 3;
[ ... ]
int liquidWidth = (int) ((SensorManager.STANDARD_
GRAVITY
  * WIDTH_MAGNIFY * 2) + bubbleDiam + 0.5);
int liquidHeight = (int) ((SensorManager.STANDARD_
GRAVITY
  * HEIGHT_MAGNIFY * 2) + bubbleDiam + 0.5);
```



› **The axes of the Android sensors.**

I think these values for the **WIDTH_MAGNIFY** and **HEIGHT_MAGNIFY** constants look best onscreen, but you can of course change them as you prefer. **SensorManager.STANDARD_GRAVITY** does exactly what you'd expect: provides the standard gravity value, which is the maximum you'll get for gravity in either direction. As well as allowing for the movement, there also needs to be room for the bubble itself; and the **0.5** ensures that any decimal value will be rounded up rather than down.

Similarly, we can use the hard-coded pixel values for the outer case of the spirit level to define the top-left x and y coordinates of the liquid and the bubble:

```
int liquidX = outerX + (outerWidth-liquidWidth)/2;
int liquidY = outerY + (outerHeight-liquidHeight)/2;
bubbleX = bubbleOrigX = outerX + (outerWidth/2) -
(bubbleDiam/2);
bubbleY = bubbleOrigY = outerY + (outerHeight/2) -
(bubbleDiam/2);
```

This will centre the liquid and the bubble in the middle of the outer case. The **moveBubble** and **onDraw** methods will then move the bubble around from there.

Recompile, install, and try it out, and you should see the little bubble moving around the screen as you tilt the phone. Add some graphics to draw a cross centred in the liquid box, and see if you can put a shelf up straight with it! LXF

Next time

Over the next few tutorials, we'll look at the Android UI, and using the touch-screen interface; graphics, screen layout, and games; handling threading in Android; and the accelerometer and other hardware sensors.

Quick tip

You can easily drop a pre-created image file into your application by saving it (as a PNG or JPG) in **res/drawable** and then referring to it with the resource ID **R.drawable.filename** (no extension).