PART III – Advanced System

CHAPTER 9

Sensors

"Sensors" is Android's overall term for ways that Android can detect elements of the physical world around it, from magnetic flux to the movement of the device. Not all devices will have all possible sensors, and other sensors are likely to be added over time. In this chapter, we will explore what sensors are theoretically available and how to use a few of them that work on early Android devices like the T-Mobile G1.

The samples in this chapter assume that you have access to a piece of sensor-equipped Android hardware, such as a T-Mobile G1. The OpenIntents.org project has a sensor simulator which you can also use, though the use of this tool is not covered here.

The author would like to thank Sean Catlin for code samples that helped clear up confusion surrounding the use of sensors.

The Sixth Sense. Or Possibly the Seventh.

In theory, Android supports the following sensor types:

- An accelerometer, that tells you the motion of the device in space through all three dimensions
- An ambient light sensor, telling you how bright or dark the surroundings are

- A magnetic field sensor, to tell you where magnetic north is (unless some other magnetic field is nearby, such as from an electrical motor)
- An orientation sensor, to tell you how the device is positioned in all three dimensions
- A proximity sensor, to tell you how far the device is from some other specific object
- A temperature sensor, to tell you the temperature of the surrounding environment
- A tricorder sensor, to turn the device into "a fully functional Tricorder"

Clearly, not all of these possible sensors are available today, such as the last one. What definitely are available today on the T-Mobile G1 are the accelerometer, the magnetic field sensor, and the orientation sensor.

To access any of these sensors, you need a `SensorManager`, found in the `android.hardware` package. Like other aspects of Android, the `SensorManager` is a system service, and as such is obtained via the `getSystemService()` method on your `Activity` or other `Context`:

```
mgr=(SensorManager)getSystemService(Context.SENSOR_SERVICE);
```

Orienting Yourself

In principle, to find out which direction is north, you would use the magnetic flux sensor and go through a lovely set of calculations to figure out the appropriate direction.

Fortunately for us, Android did all that as part of the orientation sensor...so long as the device is held flat in the horizontal plane (e.g., on a level tabletop).

Akin to the location services, there is no way to ask the `SensorManager` what the current value of a sensor is. Instead, you need to hook up a

SensorEventListener and respond to changes in the sensor values. To do this, simply call registerListener() with your SensorEventListener and the Sensor you wish to hear from. You can get the Sensor by asking the SensorManager for the default Sensor for a particular type. For example, from the Sensor/Compass sample project, here is where we register our listener:

```
mgr.registerListener(listener,
                     mgr.getDefaultSensor(Sensor.TYPE_ORIENTATION),
                     SensorManager.SENSOR_DELAY_UI);
```

Note that you also specify the rate at which sensor updates will be received. Here, we use SENSOR_DELAY_UI, but you could say SENSOR_DELAY_FASTEST or various other values.

It is important to unregister the listener when the activity closes down; otherwise, the application will never really terminate and the listener will get updates indefinitely. To do this, just call unregisterListener() from a likely location, such as onDestroy():

```
@Override
public void onDestroy() {
  super.onDestroy();
  mgr.unregisterListener(listener);
}
```

Your SensorEventListener implementation will need two methods. The one you probably will not use that often is onAccuracyChanged(), when you will be notified as a given sensor's accuracy changes from SENSOR_STATUS_ACCURACY_HIGH to SENSOR_STATUS_ACCURACY_MEDIUM to SENSOR_STATUS_ACCURACY_LOW to SENSOR_STATUS_UNRELIABLE.

The one you will use more commonly is onSensorChanged(), where you are provided a SensorEvent containing a float[] of values for the sensor. The tricky part is determining what these sensor values mean.

In the case of TYPE_ORIENTATION, the first of the supplied values represents the orientation of the device in degrees off of magnetic north. 90 degrees means east, 180 means south, and 270 means west, just like on a regular compass.

In `Sensor/Compass`, we update a `TextView` with the each reading:

```
private SensorEventListener listener=new SensorEventListener() {
  public void onSensorChanged(SensorEvent e) {
    if (e.sensor.getType()==Sensor.TYPE_ORIENTATION) {
      degrees.setText(String.valueOf(e.values[0]));
    }
  }

  public void onAccuracyChanged(Sensor sensor, int accuracy) {
    // unused
  }
};
```

What you get is a trivial application showing where the top of the phone is pointing. Note that the sensor seems to take a bit to get initially stabilized, then will tend to lag actual motion a bit.

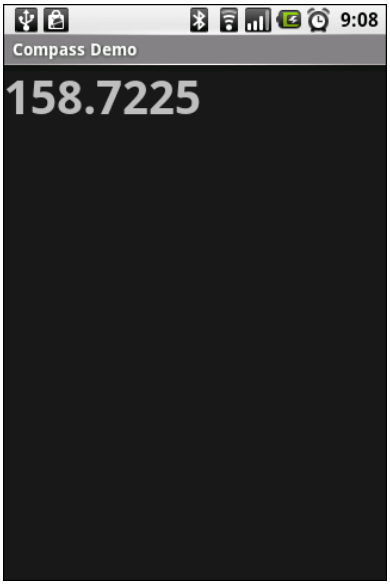

Figure 34. The CompassDemo application, showing a T-Mobile G1 pointing south-by-southeast

Steering Your Phone

In television commercials for other mobile devices, you may see them being used like a steering wheel, often times for playing a driving simulation game.

Android can do this too. You can see it in the `Sensor/Steering` sample application.

In the preceding section, we noted that `TYPE_ORIENTATION` returns in the first value of the `float[]` the orientation of the phone, compared to magnetic north, if the device is horizontal. When the device is held like a steering wheel, the second value of the `float[]` will change as the device is "steered".

This sample application is very similar to the `Sensor/Compass` one shown in the previous section. The biggest change comes in the `SensorEventListener` implementation:

```
private SensorEventListener listener=new SensorEventListener() {
  public void onSensorChanged(SensorEvent e) {
    if (e.sensor.getType()==Sensor.TYPE_ORIENTATION) {
      float orientation=e.values[1];

      if (prevOrientation!=orientation) {
        if (prevOrientation<orientation) {
          steerLeft(orientation,
                    orientation-prevOrientation);
        }
        else {
          steerRight(orientation,
                     prevOrientation-orientation);
        }

        prevOrientation=e.values[1];
      }
    }
  }

  public void onAccuracyChanged(Sensor sensor, int accuracy) {
    // unused
  }
};
```

Here, we track the previous orientation (`prevOrientation`) and call a `steerLeft()` or `steerRight()` method based on which direction the "wheel" is turned. For each, we provide the new current position of the wheel and the amount the wheel turned, measured in degrees.

The `steerLeft()` and `steerRight()` methods, in turn, simply dump their results to a "transcript": a `TextView` inside a `ScrollView`, set up to automatically keep scrolling to the bottom:

```
private void steerLeft(float position, float delta) {
  StringBuffer line=new StringBuffer("Steered left by ");

  line.append(String.valueOf(delta));
  line.append(" to ");
  line.append(String.valueOf(position));
  line.append("\n");
  transcript.setText(transcript.getText().toString()+line.toString());
  scroll.fullScroll(View.FOCUS_DOWN);
}

private void steerRight(float position, float delta) {
  StringBuffer line=new StringBuffer("Steered right by ");

  line.append(String.valueOf(delta));
  line.append(" to ");
  line.append(String.valueOf(position));
  line.append("\n");
  transcript.setText(transcript.getText().toString()+line.toString());
  scroll.fullScroll(View.FOCUS_DOWN);
}
```

The result is a log of the steering "events" as the device is turned like a steering wheel. Obviously, a real game would translate these events into game actions, such as changing your perspective of the driving course.

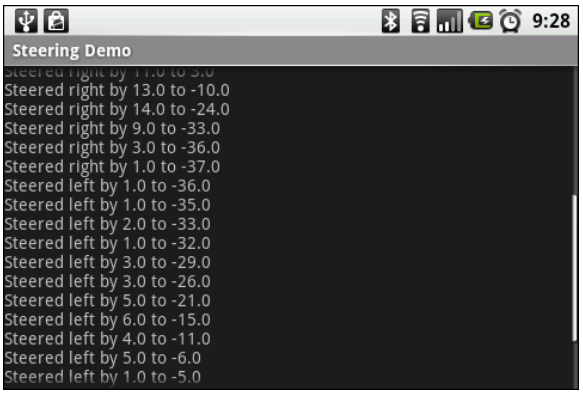

Figure 35. The SteeringDemo application

Do "The Shake"

Another demo you often see with certain other mobile devices is shaking the device to cause some on-screen effect, such as rolling dice or scrambling puzzle pieces.

Android can do this as well, as you can see in the `Sensor/Shaker` sample application, with our data provided by the accelerometer sensor (`TYPE_ACCELEROMETER`).

What the accelerometer sensor provides is the accleration in each of three dimensions. At rest, the acceleration is equal to Earth's gravity (or the gravity of wherever you are, if you are not on Earth). When shaken, the acceleration should be higher than Earth's gravity – how much higher is dependent on how hard the device is being shaken. While the individual axes of acceleration might tell you, at any point in time, what direction the device is being shaken in, since a shaking action involves frequent constant changes in direction, what we really want to know is how fast the device is moving overall – a slow steady movement is not a shake, but something more aggressive is.

Once again, our UI output is simply a "transcript" `TextView` as before. This time, though, we separate out the actual shake-detection logic into a `Shaker` class which our `ShakerDemo` activity references, as shown below:

```
package com.commonsware.android.sensor;

import android.app.Activity;
import android.os.Bundle;
import android.util.Log;
import android.view.View;
import android.widget.ScrollView;
import android.widget.TextView;

public class ShakerDemo extends Activity
  implements Shaker.Callback {
  private Shaker shaker=null;
  private TextView transcript=null;
  private ScrollView scroll=null;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    transcript=(TextView)findViewById(R.id.transcript);
    scroll=(ScrollView)findViewById(R.id.scroll);

    shaker=new Shaker(this, 1.25d, 500, this);
  }

  @Override
  public void onDestroy() {
    super.onDestroy();

    shaker.close();
  }

  public void shakingStarted() {
    Log.d("ShakerDemo", "Shaking started!");
    transcript.setText(transcript.getText().toString()+"Shaking started\n");
    scroll.fullScroll(View.FOCUS_DOWN);
  }

  public void shakingStopped() {
    Log.d("ShakerDemo", "Shaking stopped!");
    transcript.setText(transcript.getText().toString()+"Shaking stopped\n");
    scroll.fullScroll(View.FOCUS_DOWN);
  }
}
```

The `Shaker` takes four parameters:

- A `Context`, so we can get access to the `SensorManager` service
- An indication of how hard a shake should qualify as a shake, expressed as a ratio applied to Earth's gravity, so a value of `1.25`

means the shake has to be 25% stronger than gravity to be considered a shake

- An amount of time with below-threshold acceleration, after which the shake is considered "done"
- A `Shaker.Callback` object that will be notified when a shake starts and stops

While in this case, the callback methods (implemented on the `ShakerDemo` activity itself) simply log shake events to the transcript, a "real" application would, say, start an animation of dice rolling when the shake starts and end the animation shortly after the shake ends.

The `Shaker` simply converts the three individual acceleration components into a combined acceleration value (square root of the sum of the squares), then compares that value to Earth's gravity. If the ratio is higher than the supplied threshold, then we consider the device to be presently shaking, and we call the `shakingStarted()` callback method if the device was not shaking before. Once shaking ends, and time elapses, we call `shakingStopped()` on the callback object and assume that the shake has ended. A more robust implementation of `Shaker` would take into account the possibility that the sensor will not be updated for a while after the shake ends, though in reality, normal human movement will ensure that there are some sensor updates, so we can find out when the shaking ends.

```
package com.commonsware.android.sensor;

import android.content.Context;
import android.hardware.Sensor;
import android.hardware.SensorEvent;
import android.hardware.SensorEventListener;
import android.hardware.SensorManager;
import android.os.SystemClock;
import java.util.ArrayList;
import java.util.List;

public class Shaker {
  private SensorManager mgr=null;
  private long lastShakeTimestamp=0;
  private double threshold=1.0d;
  private long gap=0;
  private Shaker.Callback cb=null;
```

```
  public Shaker(Context ctxt, double threshold, long gap,
                Shaker.Callback cb) {
    this.threshold=threshold*threshold;
    this.threshold=this.threshold
                   *SensorManager.GRAVITY_EARTH
                   *SensorManager.GRAVITY_EARTH;
    this.gap=gap;
    this.cb=cb;

    mgr=(SensorManager)ctxt.getSystemService(Context.SENSOR_SERVICE);
    mgr.registerListener(listener,
                         mgr.getDefaultSensor(Sensor.TYPE_ACCELEROMETER),
                         SensorManager.SENSOR_DELAY_UI);
  }

  public void close() {
    mgr.unregisterListener(listener);
  }

  private void isShaking() {
    long now=SystemClock.uptimeMillis();

    if (lastShakeTimestamp==0) {
      lastShakeTimestamp=now;

      if (cb!=null) {
        cb.shakingStarted();
      }
    }
    else {
      lastShakeTimestamp=now;
    }
  }

  private void isNotShaking() {
    long now=SystemClock.uptimeMillis();

    if (lastShakeTimestamp>0) {
      if (now-lastShakeTimestamp>gap) {
        lastShakeTimestamp=0;

        if (cb!=null) {
          cb.shakingStopped();
        }
      }
    }
  }

  public interface Callback {
    void shakingStarted();
    void shakingStopped();
  }

  private SensorEventListener listener=new SensorEventListener() {
```

```
    public void onSensorChanged(SensorEvent e) {
      if (e.sensor.getType()==Sensor.TYPE_ACCELEROMETER) {
        double netForce=e.values[0]*e.values[0];

        netForce+=e.values[1]*e.values[1];
        netForce+=e.values[2]*e.values[2];

        if (threshold<netForce) {
          isShaking();
        }
        else {
          isNotShaking();
        }
      }
    }

    public void onAccuracyChanged(Sensor sensor, int accuracy) {
      // unused
    }
  };
}
```

All the transcript shows, of course, is when shaking starts and stops:

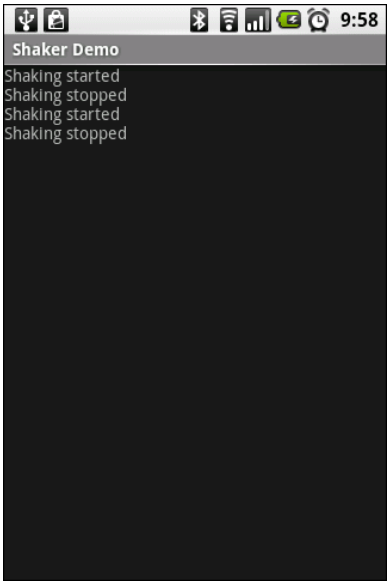

Figure 36. The ShakerDemo application, showing a pair of shakes