Chapter 2

Basics of Augmented Reality on the Android Platform

By now, you have a basic idea of what augmented reality (AR) is, what is being done with it around the world, and what you can do with it on an Android device. This chapter will launch you into the world of AR on Android and teach you the basics of it. To aid in your understanding of everything done here (and elsewhere) in this book, we will create apps that demonstrate what is being taught as we move along. This chapter will focus on making a basic app that contains the four main parts of any advanced AR app: the camera, GPS, accelerometer, and compass.

Creating the App

This is a really simple app. It has no overlays and no actual use for any of the data it is receiving from the GPS, compass, camera, and accelerometer. In the next chapter, we will build on this app and add overlays to it.

First, we need to create a new project. In the package name, I am using com.paar.ch2. You can use any name that suits you, but make sure to change any references in the code here to match your package name. The project should be set to support Android 2.1 as the minimum. I am building the project against Android 4.0 (Ice Cream Sandwich), but you can choose your own target.

Camera

The first thing in every AR app is the *camera*, which forms 99 percent of the reality in AR (the other 1 percent consists of the 3 basic sensors). To use the camera in your app, we first need to add the permission request and the `uses-feature` line to our manifest. We also must tell Android that we want our activity to be landscape and that we will handle certain config changes ourselves. After adding it, the manifest should look something like Listing 2-1:

Listing 2-1. *Updated Manifest Code*

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.paar.ch2"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk android:minSdkVersion="7" />

    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name" >
        <activity
            android:label="@string/app_name"
            android:name=".ProAndroidAR2Activity"
            android:screenOrientation = "landscape"
            android:theme="@android:style/Theme.NoTitleBar.Fullscreen"
            android:configChanges = "keyboardHidden|orientation">
            <intent-filter >
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
<uses-feature android:name="android.hardware.camera" />
<uses-permission android:name="android.permission.CAMERA" />
</manifest>
```

We can also add the permission before the start of the `<application>` element; just make sure that it is part of the manifest and is not invading into any other element.

Now let's get to the actual camera code. The camera requires a `SurfaceView`, on which it will render what it sees. We will create an XML layout with the `SurfaceView` and then use that `SurfaceView` to display the camera preview. Modify your XML file, in this case `main.xml`, to the following:

Listing 2-2. *Modified main.xml*

```
<?xml version="1.0" encoding="utf-8"?>
<android.view.SurfaceView
        xmlns:android="http://schemas.android.com/apk/res/android"
        android:id="@+id/cameraPreview"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent" >
</android.view.SurfaceView>
```

Nothing really groundbreaking in that code. Instead of using a normal layout such as `LinearLayout` or `RelativeLayout`, we simply add a `SurfaceView` to the XML file, with its height and width attributes set to allow it to fill the entire available screen. We assign it the ID `cameraPreview` so we can reference it from our code. The big step now is to use the Android camera service and tell it to tie into our `SurfaceView` to display the actual preview from the camera.

There are three things that need to be done to get this working:

1. We create a `SurfaceView`, which is in our XML layout.
2. We will also need a `SurfaceHolder`, which controls the behavior of our `SurfaceView` (for example, its size). It will also be notified when changes occur, such as when the preview starts.
3. We need a `Camera`, obtained from the `open()` static method on the `Camera` class.

To string all this together, we simply need to do the following:

4. Get the `SurfaceHolder` for our `SurfaceView` via `getHolder()`.
5. Register a `SurfaceHolder.Callback` so that we are notified when our `SurfaceView` is ready or changes.
6. Tell the `SurfaceView`, via the `SurfaceHolder`, that it has the `SURFACE_TYPE_PUSH_BUFFERS` type (using `setType()`). This indicates that something in the system will be updating the `SurfaceView` and providing the bitmap data to display.

After you've absorbed and understood all this, you can proceed to the actual coding work. First, declare the following variables, and add the imports. The top of your class should look something like this after you're done with it:

Listing 2-3. *Imports and Variable Declarations*

```
package com.paar.ch2;

import android.app.Activity;
import android.hardware.Camera;
import android.os.Bundle;
import android.util.Log;
import android.view.SurfaceHolder;
import android.view.SurfaceView;

public class ProAndroidAR2Activity extends Activity {
        SurfaceView cameraPreview;
        SurfaceHolder previewHolder;
        Camera camera;
        boolean inPreview;
```

Let me elaborate on the imports. The first and third ones are obvious, but the second one is important to note because it is for the camera. Be sure to import `Camera` from the hardware package, not the graphics package, because that is a different `Camera` class. The `SurfaceView` and `SurfaceHolder` ones are equally important, but there aren't two options to choose from.

On to the variables. `cameraPreview` is a `SurfaceView` variable that will hold the reference to the `SurfaceView` in the XML layout (this will be done in `onCreate()`). `previewHolder` is the `SurfaceHolder` to manage the `SurfaceView`. `camera` is the `Camera` object that will handle all camera stuff. Finally, `inPreview` is our little Boolean friend that will use his binary logic to tell us if a preview is active, and give us indications so that we can release it properly.

Now we move on to the `onCreate()` method for our little app:

Listing 2-4. *onCreate()*

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

        inPreview = false;

        cameraPreview = (SurfaceView)findViewById(R.id.cameraPreview);
        previewHolder = cameraPreview.getHolder();
        previewHolder.addCallback(surfaceCallback);
        previewHolder.setType(SurfaceHolder.SURFACE_TYPE_PUSH_BUFFERS);
    }
```

We set our view to our beloved `main.xml`, set `inPreview` to `false` (we are not displaying a preview of the camera right now). After that, we find our

SurfaceView from the XML file and assign it to cameraPreview. Then we run the getHolder() method, add our callback (we'll make this callback in a few minutes; don't worry about the error that will spring up right now), and set the type of previewHolder to SURFACE_TYPE_PUSH_BUFFERS.

Now a Camera object takes a setPreviewDisplay() method that takes a SurfaceHolder and arranges for the camera preview to be displayed on the related SurfaceView. However, the SurfaceView might not be ready immediately after being changed into SURFACE_TYPE_PUSH_BUFFERS mode. Therefore, although the previous setup work could be done in the onCreate() method, we should wait until the SurfaceHolder.Callback has its surfaceCreated() method called before registering the Camera. With this little explanation, we can move back to the coding:

Listing 2-5. *surfaceCallback*

```
SurfaceHolder.Callback surfaceCallback=new SurfaceHolder.Callback() {
        public void surfaceCreated(SurfaceHolder holder) {
                try {
                        camera.setPreviewDisplay(previewHolder);
                }
                catch (Throwable t) {
                   Log.e("ProAndroidAR2Activity", "Exception in
setPreviewDisplay()", t);
                }
        }
```

Now, once the SurfaceView is set up and sized by Android, we need to pass the configuration data to the Camera so it knows how big a preview it should be drawing. As Android has been ported to and installed on hundreds of different hardware devices, there is no way to safely predetermine the size of the preview pane. It would be very simple to wait for our SurfaceHolder.Callback to have its surfaceChanged() method called because this can tell us the size of the SurfaceView. Then we can push that information into a Camera.Parameters object, update the Camera with those parameters, and have the Camera show the preview via startPreview(). Now we can move back to the coding:

Listing 2-6. *sufaceChanged()*

```
public void surfaceChanged(SurfaceHolder holder, int format, int width, int
height) {
        Camera.Parameters parameters=camera.getParameters();
        Camera.Size size=getBestPreviewSize(width, height, parameters);

        if (size!=null) {
                parameters.setPreviewSize(size.width, size.height);
                camera.setParameters(parameters);
                camera.startPreview();
                inPreview=true;
        }
}
```

Eventually, you will want your app to release the camera, and reacquire it when needed. This will save resources; and many devices have only one physical camera, which can be used in only one activity at a time. There is more than one way to do this, but we will be using the `onPause()` and `onResume()` methods:

Listing 2-7. *onResume() and onPause()*

```
@Override
    public void onResume() {
      super.onResume();

      camera=Camera.open();
    }

    @Override
    public void onPause() {
      if (inPreview) {
        camera.stopPreview();
      }

      camera.release();
      camera=null;
      inPreview=false;

      super.onPause();
    }
```

You could also do it when the activity is destroyed like the following, but we will not be doing that:

Listing 2-8. *surfaceDestroyed()*

```
public void surfaceDestroyed(SurfaceHolder holder) {
                camera.stopPreview();
                camera.release();
                camera=null;
        }
```

Right about now, our little demo app should compile and display a nice little preview of what the camera sees on your screen. We aren't quite finished yet, however, because we still have to add the three sensors.

This brings us to the end of the camera part of our app. Here is the entire code for this class so far, with everything in it. You should update it to look like the following, in case you left out something:

Listing 2-9. *Full Code Listing*

```
package com.paar.ch2;

import android.app.Activity;
import android.hardware.Camera;
import android.os.Bundle;
import android.util.Log;
import android.view.SurfaceHolder;
import android.view.SurfaceView;

public class ProAndroidAR2Activity extends Activity {
        SurfaceView cameraPreview;
        SurfaceHolder previewHolder;
        Camera camera;
        boolean inPreview;
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        inPreview = false;

        cameraPreview = (SurfaceView)findViewById(R.id.cameraPreview);
        previewHolder = cameraPreview.getHolder();
        previewHolder.addCallback(surfaceCallback);
        previewHolder.setType(SurfaceHolder.SURFACE_TYPE_PUSH_BUFFERS);
    }

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

      camera=Camera.open();
    }

    @Override
    public void onPause() {
      if (inPreview) {
        camera.stopPreview();
      }

      camera.release();
      camera=null;
      inPreview=false;

      super.onPause();
    }
    private Camera.Size getBestPreviewSize(int width, int height,
Camera.Parameters parameters) {
        Camera.Size result=null;

        for (Camera.Size size : parameters.getSupportedPreviewSizes()) {
                if (size.width<=width && size.height<=height) {
                        if (result==null) {
                                result=size;
                        }
                        else {
                                int resultArea=result.width*result.height;
                                int newArea=size.width*size.height;

                                if (newArea>resultArea) {
                                        result=size;
                                }
                        }
                }
        }

        return(result);
    }

    SurfaceHolder.Callback surfaceCallback=new SurfaceHolder.Callback() {
        public void surfaceCreated(SurfaceHolder holder) {
                try {
                        camera.setPreviewDisplay(previewHolder);
                }
                catch (Throwable t) {
                        Log.e(TAG, "Exception in setPreviewDisplay()", t);
                }
        }

        public void surfaceChanged(SurfaceHolder holder, int format, int width,
int height) {
                Camera.Parameters parameters=camera.getParameters();
                Camera.Size size=getBestPreviewSize(width, height, parameters);

                if (size!=null) {
                        parameters.setPreviewSize(size.width, size.height);
                        camera.setParameters(parameters);
                        camera.startPreview();
                        inPreview=true;
                }
        }

        public void surfaceDestroyed(SurfaceHolder holder) {
                // not used
        }
    };
}
```

Orientation Sensor

The *orientation sensor* is a combination of the magnetic field sensor and the accelerometer sensor. With the data from these two sensors and a bit of trigonometry, you can get the `pitch`, `roll`, and `heading` (`azimuth`) of the device. If you like trigonometry, you'll be disappointed to know that Android does all the calculations for you, and you can simply pull the values out of a `SensorEvent`.

NOTE: Magnetic field compasses tend to go a bit crazy around metallic objects. Guess what large metallic object is likely to be close to your device while testing? Your computer! Keep that in mind if your readings aren't what you expected.

Figure 2-1 shows the axes of an orientation sensor.

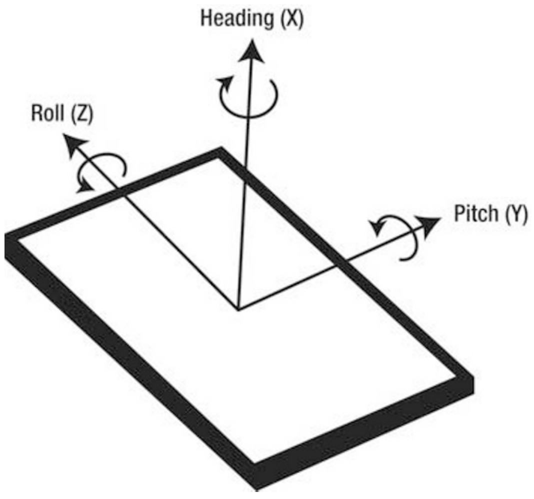


Figure 2-1. *The axes of the device.*

Before we get around to taking these values from Android and using them, let's understand a little more about what they actually are.

- **X-axis or heading**: The X-axis is a bit like a compass. It measures the direction the device is facing, where 0º or 360º is North, 90º is East, 180º is South, and 270º is West.

- **Y-axis or pitch**: This axis measures the tilt of the device. The reading will be 0º if the device is flat, -90º if the top is pointed at the ceiling, and 90º if it is upside down.
- **Z-axis or roll**: This axis measures the sideways tilt of the device. 0º is flat on its back, -90º is facing left, and 90º is the screen facing right.

There are actually two ways to get the preceding data. You can either query the orientation sensor directly, or get the readings of the accelerometer and magnetic field sensors individually and calculate the orientation. The latter is several times slower, but provides for added accuracy. In our app, we will be querying the orientation sensor directly. You can begin by adding the following variables to your class:

Listing 2-10. *New Variable Declarations*

```
final static String TAG = "PAAR";
SensorManager sensorManager;

int orientationSensor;
float headingAngle;
float pitchAngle;
float rollAngle;
```

The `TAG` string is a constant that we will use as the tag in all our log statements. The `sensorManager` will be used to get all our sensor data and to manage our sensors. The floats `headingAngle`, `pitchAngle`, and `rollAngle` will be used to store the heading, pitch and the roll of the device, respectively.

After adding the variables given above, add the following lines to your `onCreate()`:

Listing 2-11. *Implementing the SensorManager*

```
sensorManager = (SensorManager) getSystemService(SENSOR_SERVICE);
orientationSensor = Sensor.TYPE_ORIENTATION;
sensorManager.registerListener(sensorEventListener,
    sensorManager.getDefaultSensor(orientationSensor),
    SensorManager.SENSOR_DELAY_NORMAL);
```

`SensorManager` is a system service, and we get a reference to it in the first line. We then assign to `orientationSensor` the constant value of `Sensor.TYPE_ORIENTATION`, which is basically the constant given to the orientation sensor. Finally, we register our `SensorEventListener` for the default orientation sensor, with the normal delay. `SENSOR_DELAY_NORMAL` is suitable for UI changes, `SENSOR_DELAY_GAME` is suitable for use in games, `SENSOR_DELAY_UI` is suitable for updating the UI thread, and `SENSOR_DELAY_FASTEST` is the fastest the hardware supports. These settings tell Android approximately how often you want updates from the sensor. Android will not always give it at exactly the intervals s pecified. I t m ay return v alues a l ittle s lower or f aster—generally faster. You should only use the delay that you need because sensors consume a lot of CPU and battery life.

Right about now, there should be a red underline under `sensorEventListener`. This is because we haven't actually created the listener so far; we will do that now:

Listing 2-12. *sensorEventListener*

```
final SensorEventListener sensorEventListener = new SensorEventListener() {
    public void onSensorChanged(SensorEvent sensorEvent) {
            if (sensorEvent.sensor.getType() == Sensor.TYPE_ORIENTATION)
            {
                    headingAngle = sensorEvent.values[0];
                    pitchAngle = sensorEvent.values[1];
                    rollAngle = sensorEvent.values[2];

                    Log.d(TAG, "Heading: " + String.valueOf(headingAngle));
                    Log.d(TAG, "Pitch: " + String.valueOf(pitchAngle));
                    Log.d(TAG, "Roll: " + String.valueOf(rollAngle));
            }
    }

    public void onAccuracyChanged (Sensor senor, int accuracy) {
            //Not used
    }
};
```

We create and register `sensorEventListener` as a new `SensorEventListener`. We then use the `onSensorChanged()` method to receive updates when the values of the sensors change. Because `onSensorChanged()` receives updates for all sensors, we use an `if` statement to filter out everything except the orientation sensor. We then store the values from the sensor in our variables, and print them out to the log. We could also overlay this data on the camera preview, but that is beyond the scope of this chapter. We also have the `onAccuracyChanged()` method present, which we aren't using for now. It's just there because you must implement it, according to Eclipse.

Now so that our app behaves nicely and doesn't kill off the user's battery, we will register and unregister our sensor in the `onResume()` and `onPause()` methods. Update them to the following:

Listing 2-13. *onResume() and onPause()*

```
@Override
public void onResume() {
  super.onResume();
  sensorManager.registerListener(sensorEventListener, sensorManager
.getDefaultSensor(orientationSensor), SensorManager.SENSOR_DELAY_NORMAL);
  camera=Camera.open();
}

@Override
public void onPause() {
  if (inPreview) {
    camera.stopPreview();
  }
  sensorManager.unregisterListener(sensorEventListener);
  camera.release();
  camera=null;
  inPreview=false;

  super.onPause();
}
```

This wraps up the section on the orientation sensor. We'll now take a look at the accelerometer sensor.

Accelerometer

The *accelerometer* measures acceleration along three directional axes: left-right (lateral(X)), forward-backward (longitudinal(Y)) and up-down (vertical(Z)). These values are passed along in the float array of value.

Figure 2-2 shows the axes of the accelerometer.

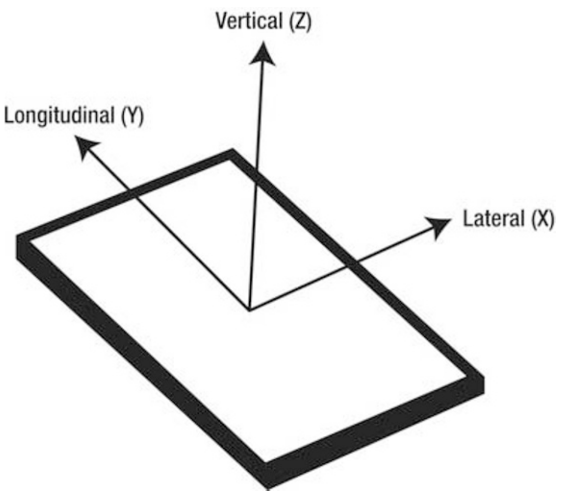


Figure 2-2. *Accelerometer axes*

In our application, we will be receiving the accelerometer values and outputting them through the LogCat. Later on in the book, we will use the accelerometer to determine speed and other things.

Let's take a very quick look at the axes of the accelerometer and exactly what they measure.

- X-Axis: On a normal device with a normal accelerometer, the X-axis measures lateral acceleration. That is, left to right; right to left. The reading is positive if you are moving it to your right side, and is negative if you are moving it to your left. For example, a device flat on its back, facing up, and in portrait orientation being moved along a surface to your right will generate a positive reading on the X-axis.
- Y-Axis: The Y-axis functions the same way as the X-axis, except it measures the acceleration longitudinally. A positive reading is registered when a device held in the same configuration described in the X-axis is moved in the direction of its top, and a negative reading is registered if moved in the opposite direction.
- Z-Axis: This axis measures the acceleration for upward and downward motion, for which positive readings are upward motions, and negative readings are downward motions. When at rest, you will get a reading of approximately -9.8m/s^2 due to gravity. In your calculations, this should be accounted for.

Let's start with the coding work now. We will be using the same `SensorManager` as before with the accelerometer. We will simply need to add a few variables, get the accelerometer sensor, and add another filtering `if` statement in the `onSensorChanged()` method. Let's start with the variables:

Listing 2-14. *Accelerometer Variables*

```
int accelerometerSensor;
float xAxis;
float yAxis;
float zAxis;
```

`accelerometerSensor` will be used to store the constant for the accelerometer, `xAxis` will store the value returned by the sensor for the X-axis, `yAxis` will store the value returned by the sensor for the Y-axis, and `zAxis` will store the value returned by the sensor for the Z-axis.

After adding the variables, we will need to update our sensor-related code in the `onCreate()` method as well, so that we can use and listen for the accelerometer later on in the `onSensorChanged()` method. Modify the sensor code in the `onCreate()` to the following:

Listing 2-15. *Modified onCreate()*

```
sensorManager = (SensorManager) getSystemService(SENSOR_SERVICE);

        orientationSensor = Sensor.TYPE_ORIENTATION;
        accelerometerSensor = Sensor.TYPE_ACCELEROMETER;

        sensorManager.registerListener(sensorEventListener, sensorManager
.getDefaultSensor(orientationSensor), SensorManager.SENSOR_DELAY_NORMAL);

        sensorManager.registerListener(sensorEventListener, sensorManager
.getDefaultSensor(accelerometerSensor), SensorManager.SENSOR_DELAY_NORMAL);
```

We have simply repeated for the accelerometer what we had already done for the orientation sensor, so you should have no problem understanding what is going on here. Now we must update the `sensorEventListener` to listen for the accelerometer by changing the code to the following:

Listing 2-16. *Modified sensorEventListener()*

```
final SensorEventListener sensorEventListener = new SensorEventListener() {
    public void onSensorChanged(SensorEvent sensorEvent) {
            if (sensorEvent.sensor.getType() == Sensor.TYPE_ORIENTATION)
            {
                    headingAngle = sensorEvent.values[0];
                    pitchAngle = sensorEvent.values[1];
```

```
                rollAngle = sensorEvent.values[2];

                Log.d(TAG, "Heading: " + String.valueOf(headingAngle));
                Log.d(TAG, "Pitch: " + String.valueOf(pitchAngle));
                Log.d(TAG, "Roll: " + String.valueOf(rollAngle));
        }

        else if (sensorEvent.sensor.getType() == Sensor.TYPE_ACCELEROMETER)
        {
                xAxis = sensorEvent.values[0];
                yAxis = sensorEvent.values[1];
                zAxis = sensorEvent.values[2];

                Log.d(TAG, "X Axis: " + String.valueOf(xAxis));
                Log.d(TAG, "Y Axis: " + String.valueOf(yAxis));
                Log.d(TAG, "Z Axis: " + String.valueOf(zAxis));

        }
    }
```

Again, we are repeating what we did for the orientation sensor to listen to the accelerometer sensor changes. We use `if` statements to distinguish between the two sensors, update the appropriate floats with the new values, and print the new values out to the log. Now all that remains is to update the `onResume()` method to register the accelerometer again:

Listing 2-17. *Modified onResume()*

```
@Override
public void onResume() {
  super.onResume();

  sensorManager.registerListener(sensorEventListener, sensorManager
.getDefaultSensor(orientationSensor), SensorManager.SENSOR_DELAY_NORMAL);

  sensorManager.registerListener(sensorEventListener, sensorManager
.getDefaultSensor(accelerometerSensor), SensorManager.SENSOR_DELAY_NORMAL);

  camera=Camera.open();
}
```

We do not need to change anything in `onPause()` as we unregister the entire listener there, all associated sensors included.

With that, we come to the end of our two sensors. Now all that is left to complete our app is to implement the GPS.

Global Positioning System (GPS)

The *global positioning system (GPS)* is a location system that can give an extremely accurate location via satellites. It will be the final part of our amazing little demo app.

First, let's take a brief look at the history of the GPS and how it works.

The GPS is a space-based satellite navigation system. It is managed by the United States and is available for use by anyone with a GPS receiver, although it was originally intended to be military only.

Originally, there were 24 satellites to which a receiver would communicate. The system has been upgraded over the years to have 31 satellites, plus 2 older ones that are currently marked as spares. At any time, a minimum of nine satellites can be viewed from the ground, while the rest are not visible.

To obtain a fix, a receiver must communicate with a minimum of four satellites. The satellites send three pieces of information to the receiver, which are then fed into one of the many algorithms for finding the actual location. The three pieces are the time of broadcast, the orbital location of that particular satellite, and the rough locations of all the other satellites (system health or almanac). The location is calculated using trigonometry. This may make you think that in such a case, three satellites will be enough to obtain a fix, but a timing error in the communications, when multiplied by the speed of light that is used in the algorithms, results in a very big error in the final location.

For our sensor data, we used a `SensorManager`. To use the GPS, however, we will be using a `LocationManager`. Although we used a `SensorEventListener` for the sensors, we will use a `LocationListener` for the GPS. To start off, we will declare the variables that we will be using:

Listing 2-18. *Declaring GPS Variables*

```
LocationManager locationManager;
double latitude;
double longitude;
double altitude;
```

We will only be taking the latitude, longitude and altitude from our `Location` object, but you can also get the bearing, time, and so forth if you want. It all depends on what you want your app to do, and what data you need to do it. Before we get around to actually getting all this data, let's take a look at what latitude and longitude are.

Latitude and Longitude

Latitudes are part of the Earth's grid system; they are imaginary circles that go from the North Pole to the South Pole. The equator is the 0° line, and the only one of the latitudes that is a great circle. All latitudes are parallel to one another. Each latitude is approximately 69 miles, or 111 kilometers, from its immediate previous and next ones. The exact distance varies due to the curvature of the Earth.

Figure 2-3 shows the concept of a sphere.

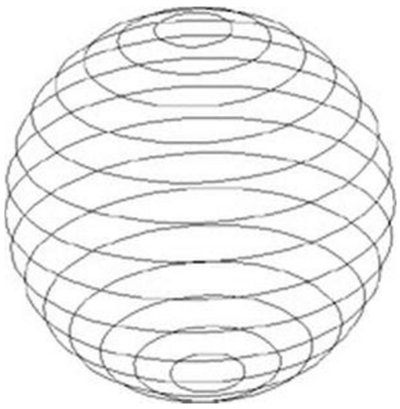


Figure 2-3. *A graphical representation of latitudes*

Longitudes are also imaginary lines of the Earth's grid system. They run from the North Pole to the South Pole, converging at each of the poles. Each longitude is half of a great circle. The 0° longitude is known as the Prime Meridian and passes through Greenwich, England. The distance between two longitudes is greatest at the equator, and is approximately 69 miles, or 111 kilometers, the same as the approximate distance between two latitudes.

Figure 2-4 shows the concept on another sphere.

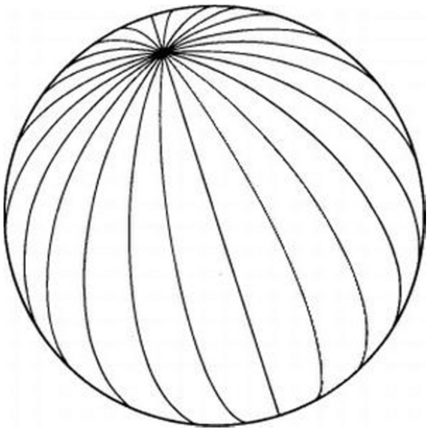


Figure 2-4. *A graphical representation of longitudes*

With a new understanding of latitudes and longitudes, we can move on to getting the service from the system and asking for location updates in the `onCreate()` method:

Listing 2-19. *Asking for Location Updates in onCreate()*

```
locationManager = (LocationManager) getSystemService(LOCATION_SERVICE);
locationManager.requestLocationUpdates(LocationManager.GPS_PROVIDER, 2000, 2,
 locationListener);
```

First, we get the location service from Android. After that, we use the `requestLocationUpdates()` method to request the location updates. The first parameter is the constant of the provider we want to use (in this case, the GPS). We can also use the cell network. The second parameter is the time interval between updates in milliseconds, the third is the minimum distance that the device should move in meters, and the last parameter is the `LocationListener` that should be notified.

Right now, the `locationListener` should have a red underline. That is because we haven't yet quite made it. Let's fix that:

Listing 2-20. *locationListener*

```
LocationListener locationListener = new LocationListener() {
    public void onLocationChanged(Location location) {
            latitude = location.getLatitude();
            longitude = location.getLongitude();
            altitude = location.getAltitude();

            Log.d(TAG, "Latitude: " + String.valueOf(latitude));
            Log.d(TAG, "Longitude: " + String.valueOf(longitude));
            Log.d(TAG, "Altitude: " + String.valueOf(altitude));
    }

            public void onProviderDisabled(String arg0) {
                    // TODO Auto-generated method stub

            }

            public void onProviderEnabled(String arg0) {
                    // TODO Auto-generated method stub

            }

            public void onStatusChanged(String arg0, int arg1, Bundle arg2) {
                    // TODO Auto-generated method stub

            }
};
```

The onLocationChanged() method is invoked every time your minimum time interval takes place or the device moves the minimum distance you specified or more. The Location object received by the method contains a whole host of information: the latitude, longitude, altitude, bearing, and so on. However, in this example we extract and save only the latitude, altitude, and longitude. The Log.d statements simply display the values received.

The GPS is one of the most battery-intensive parts of the Android system and could drain out a fully charged battery in a few hours. This is why we will go through the whole thing of release and acquiring the GPS in the onPause() and onResume() methods:

Listing 2-21. *onResume() and onPause()*

```
@Override
    public void onResume() {
      super.onResume();
      locationManager.requestLocationUpdates(LocationManager.GPS_PROVIDER, 2000,
2, locationListener);
      sensorManager.registerListener(sensorEventListener, sensorManager
.getDefaultSensor(orientationSensor), SensorManager.SENSOR_DELAY_NORMAL);
      sensorManager.registerListener(sensorEventListener, sensorManager
.getDefaultSensor(accelerometerSensor), SensorManager.SENSOR_DELAY_NORMAL);
      camera=Camera.open();
    }

    @Override
    public void onPause() {
      if (inPreview) {
            camera.stopPreview();
        }
        locationManager.removeUpdates(locationListener);
        sensorManager.unregisterListener(sensorEventListener);
        camera.release();
        camera=null;
        inPreview=false;

        super.onPause();
    }
```

This brings us to the end of our demo app. If done right, you should see the camera preview on the screen, coupled with a fast moving LogCat. All the files modified from the default state at project creation are given here now, so that you can make sure that everything is in place.

ProAndroidAR2Activity.java

Listing 2-22. *Full listing for ProAndroidAR2Activity.java*

```
package com.paar.ch2;

import android.app.Activity;
import android.hardware.Camera;
import android.hardware.Sensor;
import android.hardware.SensorEvent;
import android.hardware.SensorEventListener;
import android.hardware.SensorManager;
import android.location.Location;
import android.location.LocationListener;
import android.location.LocationManager;
import android.os.Bundle;
import android.util.Log;
import android.view.SurfaceHolder;
import android.view.SurfaceView;

public class ProAndroidAR2Activity extends Activity{
        SurfaceView cameraPreview;
        SurfaceHolder previewHolder;
        Camera camera;
        boolean inPreview;

        final static String TAG = "PAAR";
        SensorManager sensorManager;

        int orientationSensor;
        float headingAngle;
        float pitchAngle;
        float rollAngle;

        int accelerometerSensor;
        float xAxis;
        float yAxis;
        float zAxis;

        LocationManager locationManager;
        double latitude;
        double longitude;
        double altitude;
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        locationManager = (LocationManager) getSystemService(LOCATION_SERVICE);
        locationManager.requestLocationUpdates(LocationManager.GPS_PROVIDER,
2000, 2, locationListener);

        sensorManager = (SensorManager) getSystemService(SENSOR_SERVICE);
        orientationSensor = Sensor.TYPE_ORIENTATION;
        accelerometerSensor = Sensor.TYPE_ACCELEROMETER;
        sensorManager.registerListener(sensorEventListener, sensorManager
.getDefaultSensor(orientationSensor), SensorManager.SENSOR_DELAY_NORMAL);
        sensorManager.registerListener(sensorEventListener, sensorManager
.getDefaultSensor(accelerometerSensor), SensorManager.SENSOR_DELAY_NORMAL);

        inPreview = false;

        cameraPreview = (SurfaceView)findViewById(R.id.cameraPreview);
        previewHolder = cameraPreview.getHolder();
        previewHolder.addCallback(surfaceCallback);
        previewHolder.setType(SurfaceHolder.SURFACE_TYPE_PUSH_BUFFERS);
    }

    LocationListener locationListener = new LocationListener() {
        public void onLocationChanged(Location location) {
                latitude = location.getLatitude();
                longitude = location.getLongitude();
                altitude = location.getAltitude();

                Log.d(TAG, "Latitude: " + String.valueOf(latitude));
                Log.d(TAG, "Longitude: " + String.valueOf(longitude));
                Log.d(TAG, "Altitude: " + String.valueOf(altitude));
        }

                public void onProviderDisabled(String arg0) {
                        // TODO Auto-generated method stub
                }

                public void onProviderEnabled(String arg0) {
                        // TODO Auto-generated method stub

                }

                public void onStatusChanged(String arg0, int arg1, Bundle arg2)
{
                        // TODO Auto-generated method stub

                }
    };

    final SensorEventListener sensorEventListener = new SensorEventListener() {
        public void onSensorChanged(SensorEvent sensorEvent) {
                if (sensorEvent.sensor.getType() == Sensor.TYPE_ORIENTATION)
                {
                        headingAngle = sensorEvent.values[0];
                        pitchAngle = sensorEvent.values[1];
                        rollAngle = sensorEvent.values[2];

                        Log.d(TAG, "Heading: " + String.valueOf(headingAngle));
                        Log.d(TAG, "Pitch: " + String.valueOf(pitchAngle));
                        Log.d(TAG, "Roll: " + String.valueOf(rollAngle));
                }

                else if (sensorEvent.sensor.getType() ==
Sensor.TYPE_ACCELEROMETER)
                {
                        xAxis = sensorEvent.values[0];
                        yAxis = sensorEvent.values[1];
                        zAxis = sensorEvent.values[2];

                        Log.d(TAG, "X Axis: " + String.valueOf(xAxis));
                        Log.d(TAG, "Y Axis: " + String.valueOf(yAxis));
                        Log.d(TAG, "Z Axis: " + String.valueOf(zAxis));

                }
        }

        public void onAccuracyChanged (Sensor senor, int accuracy) {
                //Not used
        }
    };

    @Override
    public void onResume() {
      super.onResume();
      locationManager.requestLocationUpdates(LocationManager.GPS_PROVIDER, 2000,
2, locationListener);
      sensorManager.registerListener(sensorEventListener, sensorManager
.getDefaultSensor(orientationSensor), SensorManager.SENSOR_DELAY_NORMAL);
      sensorManager.registerListener(sensorEventListener, sensorManager
.getDefaultSensor(accelerometerSensor), SensorManager.SENSOR_DELAY_NORMAL);
      camera=Camera.open();
    }

    @Override
    public void onPause() {
      if (inPreview) {
        camera.stopPreview();
      }
      locationManager.removeUpdates(locationListener);
      sensorManager.unregisterListener(sensorEventListener);
      camera.release();
      camera=null;
      inPreview=false;

      super.onPause();
    }

    private Camera.Size getBestPreviewSize(int width, int height,
Camera.Parameters parameters) {
        Camera.Size result=null;

        for (Camera.Size size : parameters.getSupportedPreviewSizes()) {
                if (size.width<=width && size.height<=height) {
                        if (result==null) {
                                result=size;
                        }
                        else {
                                int resultArea=result.width*result.height;
                                int newArea=size.width*size.height;

                                if (newArea>resultArea) {
                                        result=size;
                                }
                        }
                }
        }

        return(result);
    }

    SurfaceHolder.Callback surfaceCallback=new SurfaceHolder.Callback() {
        public void surfaceCreated(SurfaceHolder holder) {
                try {
                        camera.setPreviewDisplay(previewHolder);
                }
                catch (Throwable t) {
                        Log.e(TAG, "Exception in setPreviewDisplay()", t);
                }
        }

        public void surfaceChanged(SurfaceHolder holder, int format, int width,
int height) {
                Camera.Parameters parameters=camera.getParameters();
                Camera.Size size=getBestPreviewSize(width, height, parameters);

                if (size!=null) {
                        parameters.setPreviewSize(size.width, size.height);
                        camera.setParameters(parameters);
                        camera.startPreview();
                        inPreview=true;
                }
        }

        public void surfaceDestroyed(SurfaceHolder holder) {
                // not used
        }

    };
}
```

AndroidManifest.xml

Listing 2-23. *Full listing for AndroidManifest.xml*

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.paar.ch2"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk android:minSdkVersion="7" />

    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name" >
        <activity
            android:label="@string/app_name"
            android:name=".ProAndroidAR2Activity"
            android:screenOrientation = "landscape"
            android:theme="@android:style/Theme.NoTitleBar.Fullscreen"
            android:configChanges = "keyboardHidden|orientation">
            <intent-filter >
```

```
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
<uses-feature android:name="android.hardware.camera" />
<uses-permission android:name="android.permission.CAMERA" />
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
</manifest>
```

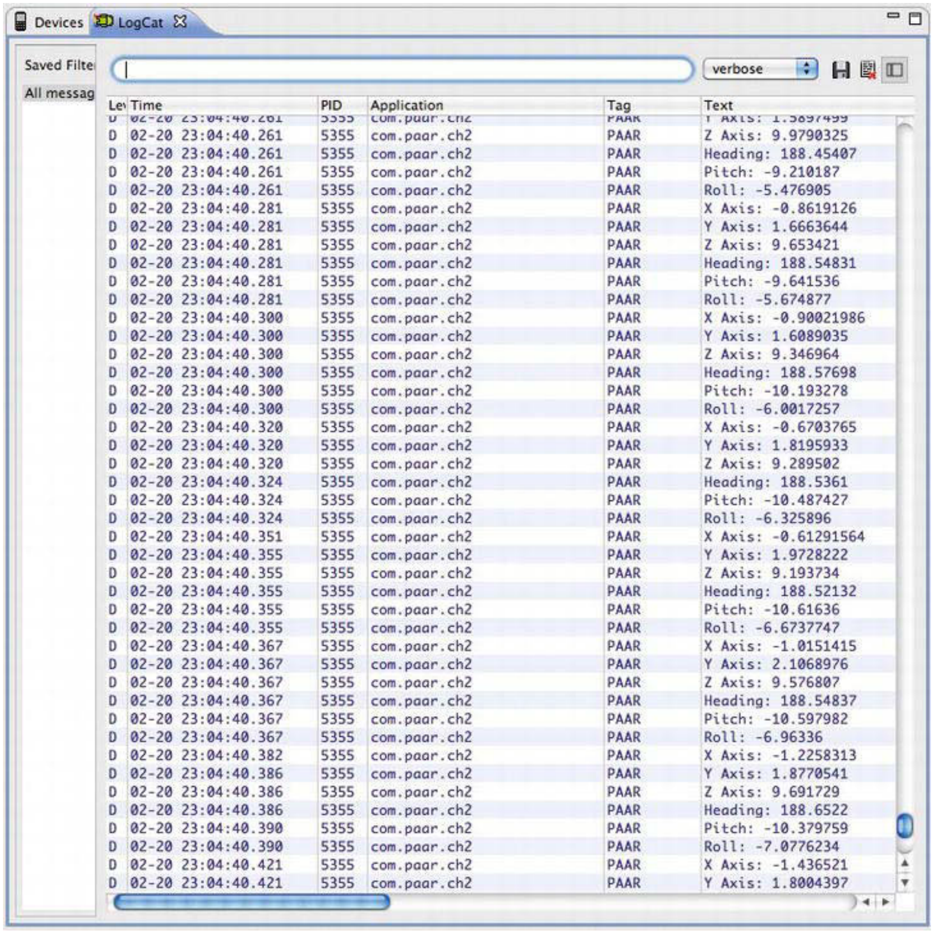
main.xml

Listing 2-24. *Full listing for main.xml*

```
<?xml version="1.0" encoding="utf-8"?>
<android.view.SurfaceView
        xmlns:android="http://schemas.android.com/apk/res/android"
        android:id="@+id/cameraPreview"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent" >
</android.view.SurfaceView>
```

Sample LogCat Output

After you have written the app out, run it from Eclipse using the Run As button on top. If you are running it on an emulator, you will get nothing because the sensors are not emulated. On a device, you should see a camera preview on the screen, coupled with a fast-moving LogCat that looks something like Figure 2-5.



```
Le Time                 PID   Application    Tag    Text
D  02-20 23:04:40.261   5355  com.paar.ch2   PAAR   Y Axis: 1.5897499
D  02-20 23:04:40.261   5355  com.paar.ch2   PAAR   Z Axis: 9.9790325
D  02-20 23:04:40.261   5355  com.paar.ch2   PAAR   Heading: 188.45407
D  02-20 23:04:40.261   5355  com.paar.ch2   PAAR   Pitch: -9.210187
D  02-20 23:04:40.261   5355  com.paar.ch2   PAAR   Roll: -5.476905
D  02-20 23:04:40.281   5355  com.paar.ch2   PAAR   X Axis: -0.8619126
D  02-20 23:04:40.281   5355  com.paar.ch2   PAAR   Y Axis: 1.6663644
D  02-20 23:04:40.281   5355  com.paar.ch2   PAAR   Z Axis: 9.653421
D  02-20 23:04:40.281   5355  com.paar.ch2   PAAR   Heading: 188.54831
D  02-20 23:04:40.281   5355  com.paar.ch2   PAAR   Pitch: -9.641536
D  02-20 23:04:40.281   5355  com.paar.ch2   PAAR   Roll: -5.674877
D  02-20 23:04:40.300   5355  com.paar.ch2   PAAR   X Axis: -0.90021986
D  02-20 23:04:40.300   5355  com.paar.ch2   PAAR   Y Axis: 1.6089035
D  02-20 23:04:40.300   5355  com.paar.ch2   PAAR   Z Axis: 9.346964
D  02-20 23:04:40.300   5355  com.paar.ch2   PAAR   Heading: 188.57698
D  02-20 23:04:40.300   5355  com.paar.ch2   PAAR   Pitch: -10.193278
D  02-20 23:04:40.300   5355  com.paar.ch2   PAAR   Roll: -6.0017257
D  02-20 23:04:40.320   5355  com.paar.ch2   PAAR   X Axis: -0.6703765
D  02-20 23:04:40.320   5355  com.paar.ch2   PAAR   Y Axis: 1.8195933
D  02-20 23:04:40.320   5355  com.paar.ch2   PAAR   Z Axis: 9.289502
D  02-20 23:04:40.324   5355  com.paar.ch2   PAAR   Heading: 188.5361
D  02-20 23:04:40.324   5355  com.paar.ch2   PAAR   Pitch: -10.487427
D  02-20 23:04:40.324   5355  com.paar.ch2   PAAR   Roll: -6.325896
D  02-20 23:04:40.351   5355  com.paar.ch2   PAAR   X Axis: -0.61291564
D  02-20 23:04:40.355   5355  com.paar.ch2   PAAR   Y Axis: 1.9728222
D  02-20 23:04:40.355   5355  com.paar.ch2   PAAR   Z Axis: 9.193734
D  02-20 23:04:40.355   5355  com.paar.ch2   PAAR   Heading: 188.52132
D  02-20 23:04:40.355   5355  com.paar.ch2   PAAR   Pitch: -10.61636
D  02-20 23:04:40.355   5355  com.paar.ch2   PAAR   Roll: -6.6737747
D  02-20 23:04:40.367   5355  com.paar.ch2   PAAR   X Axis: -1.0151415
D  02-20 23:04:40.367   5355  com.paar.ch2   PAAR   Y Axis: 2.1068976
D  02-20 23:04:40.367   5355  com.paar.ch2   PAAR   Z Axis: 9.576807
D  02-20 23:04:40.367   5355  com.paar.ch2   PAAR   Heading: 188.54837
D  02-20 23:04:40.367   5355  com.paar.ch2   PAAR   Pitch: -10.597982
D  02-20 23:04:40.367   5355  com.paar.ch2   PAAR   Roll: -6.96336
D  02-20 23:04:40.382   5355  com.paar.ch2   PAAR   X Axis: -1.2258313
D  02-20 23:04:40.386   5355  com.paar.ch2   PAAR   Y Axis: 1.8770541
D  02-20 23:04:40.386   5355  com.paar.ch2   PAAR   Z Axis: 9.691729
D  02-20 23:04:40.386   5355  com.paar.ch2   PAAR   Heading: 188.6522
D  02-20 23:04:40.390   5355  com.paar.ch2   PAAR   Pitch: -10.379759
D  02-20 23:04:40.390   5355  com.paar.ch2   PAAR   Roll: -7.0776234
D  02-20 23:04:40.421   5355  com.paar.ch2   PAAR   X Axis: -1.436521
D  02-20 23:04:40.421   5355  com.paar.ch2   PAAR   Y Axis: 1.8004397
```

Figure 2-5. *Screenshot of the LogCat while the app is running.*

If you have a clear view of the sky, the LogCat will also include three lines that tell you the latitude, longitude, and altitude.

Summary

In this chapter, you learned how to use the camera, how to read the values from the accelerometer and orientation sensor, and how to get the user's location using GPS.

You also learned to utilize the four base components of any full-featured AR app. You will not always use all four of these things in your app. In fact, it is very rare to have an app with such a requirement.

This chapter should give you a basic understanding of AR, and the project from this app is essentially the skeleton of a proper AR app.

The next chapter discusses overlays and how they give the user a real augmented experience.