Chapter 3

Adding Overlays

As mentioned before, augmented reality (AR) is the overlaying of data that is related to the direct or indirect camera preview being displayed. In the majority of AR apps, the camera preview is first scanned for markers. In the translator kind of apps, the preview is scanned for text. And in some gaming apps, no scanning is done; instead, characters, buttons, text, and so on are overlaid on the preview.

All the source code from this chapter can be downloaded from this book's page at `http://www.apress.com` or the GitHub repository.

In Chapter 2, we made a basic app that displayed the device camera's preview, retrieved the location via GPS, got the accelerometer readings, and retrieved the orientation sensor readings. We will keep building on this app in this chapter and also add overlays to it. We will be adding normal Android widget overlays and implementing marker recognition. Let's start with the simplest: widget overlays.

Widget Overlays

The Android platform provides a bunch of standard widgets such as `TextViews`, `Buttons`, and `Checkboxes`. These are included by default in the Android OS and can be used by any application. They are probably the easiest things you can overlay on your camera preview.

To get started, create a new Android project. The one used in this example is called Pro Android AR 3 Widget Overlay, builds against Android 4, has its minSdkVersion set to 7 (Android 2.1), and has the package name com.paar.ch3widgetoverlay. (You can change all that to whatever suits your

needs, but be sure to also update the example code given here.) Figure 3-1 shows the project setup screen.

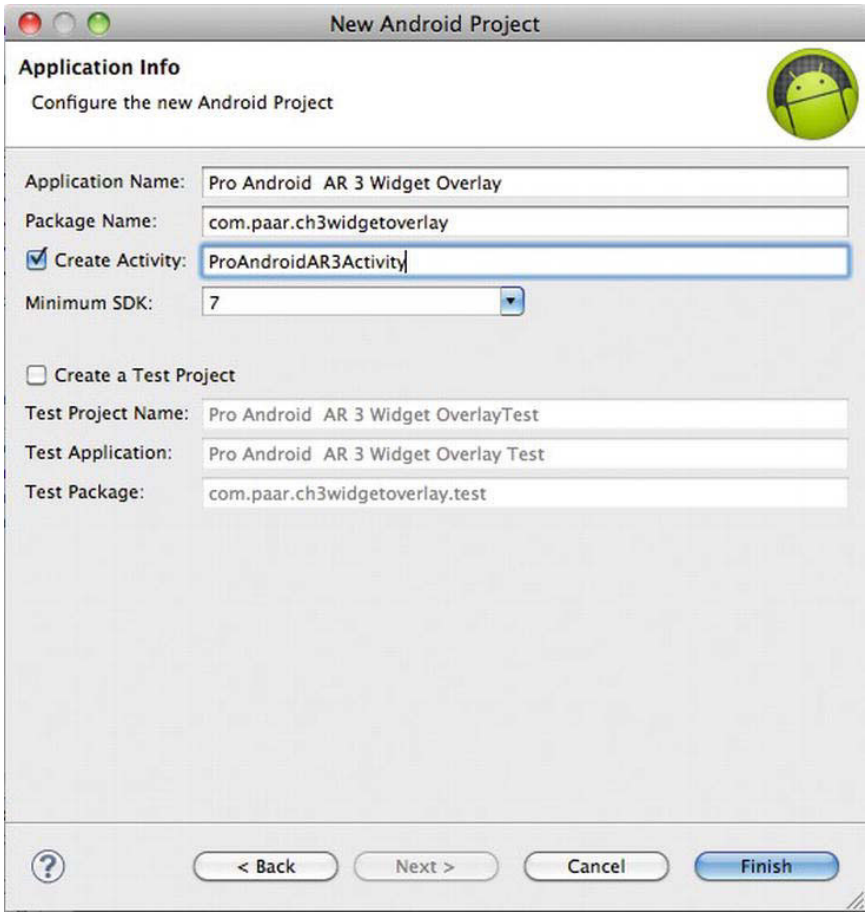

Figure 3-1. *The application details*

Start by duplicating everything we did in the last chapter. You can type it all out by hand, or copy and paste. That's your call. Be sure to update the package name and so on in the code so that it works in the new project.

After everything from the previous chapter's code is duplicated, we will need to modify the XML file defining our layout to allow for widget overlays. Earlier, the entire layout was a single `SurfaceView`, which displayed the camera preview. Because of this, we cannot currently add other widgets to the layout. Therefore, we will change our XML file to have a `RelativeLayout`, and have the `SurfaceView`

and all other widgets inside that RelativeLayout. We are using a RelativeLayout because it allows us to easily overlap the widgets on the SurfaceView. In this example, we will be adding various TextViews to display the sensor data. So before we get to the layout editing, we need to add some string resources to the project's strings.xml:

Listing 3-1. *String Resources*

```
<string name="xAxis">X Axis:</string>
<string name="yAxis">Y Axis:</string>
<string name="zAxis">Z Axis:</string>
<string name="heading">Heading:</string>
<string name="pitch">Pitch:</string>
<string name="roll">Roll:</string>
<string name="altitude">Altitude:</string>
<string name="longitude">Longitude:</string>
<string name="latitude">Latitude:</string>
<string name="empty"></string>
```

These strings will provide the labels for some of the TextViews. Half of them, to be exact. The other half will be updated with the data from the sensors. After this, you should update your main.xml file so that it has a RelativeLayout. Let's take a quick look at what a RelativeLayout is and how it compares to other layouts before we move onto the actual code.

Layout Options

In Android, there are many different root layouts available. These layouts define the user interface of any app. All layouts are usually defined in XML files located in /res/layout. However, layouts and their elements can be dynamically created at runtime through Java code. This is done only if the app needs to add widgets on the fly. Layouts can be declared in XML and then be modified later through the Java code, as we will frequently do in our apps. This is done by acquiring a reference to a part of the layout, for example a TextView, and calling various methods of that class to alter it. We can do this because each layout element (including the layout) has a corresponding Java class in the Android framework, which defines the methods that modify it. There are currently four different layout options:

- Frame layout
- Table layout
- Linear layout
- Relative layout

When Android was first released, there was a fifth layout option called Absolute layout. This layout allowed you to specify the position of an element using exact x and y coordinates. This layout is now deprecated because it is difficult to use across different screen sizes.

Frame Layout

The Frame layout is the simplest type of layout. It is essentially a large blank space on which you can put a single child object, which will be pinned to the top-left corner of the screen. Any other objects added after the first one will be drawn directly on top of it.

Table Layout

A Table layout positions its children into rows and columns. `TableLayout` containers do not display border lines for their rows, columns, or cells. The table will have as many columns as the row with the most cells. A table can leave cells empty, but cells can't span columns as they can in HTML. `TableRow` objects are the child views of a `TableLayout` (each `TableRow` defines a single row in the table). Each row has zero or more cells, each of which is defined by any kind of other view. So the cells of a row can be composed of a variety of View objects such as `ImageView` or `TextView`. A cell can also be a `ViewGroup` object (for example, you can nest another `TableLayout` as a cell).

Linear Layout

A Linear layout aligns all children in a single direction—vertically or horizontally, depending on how you define the orientation attribute. All children are stacked one after the other, so a vertical list has only one child per row, no matter how wide they are; and a horizontal list is only one row high (the height of the tallest child, plus padding). A `LinearLayout` respects margins between children and the gravity (right, center, or left alignment) of each child.

Relative Layout

Finally, the Relative layout lets child views specify their position relative to the parent view or to each other (specified by ID). So you can align two elements by the right border, make one below another, center it in the screen, center it left, and so on. Elements are rendered in the order given, so if the first element is centered in the screen, other elements aligning themselves to that element will

be aligned relative to the screen center. Also, because of this ordering, if using XML to specify this layout, the element that you will reference (in order to position other view objects) must be listed in the XML file before you refer to it from the other views via its reference ID.

A Relative layout is the only layout that allows us to overlap views in the way that our application needs it. Due to its need to reference other parts of the layout to place a view on the screen, you must ensure that all `RelativeLayouts` in this book are copied out exactly into your code; otherwise, your entire layout will look very jumbled up, with views going everywhere except where you put them.

Updating main.xml with a RelativeLayout

Inside the `RelativeLayout` is one `SurfaceView` and 18 `TextViews`. The ordering and IDs of the widgets are important because it is a `RelativeLayout`. Here is the layout file:

Listing 3-2. *RelativeLayout*

```
<RelativeLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
        android:id="@+id/relativeLayout1"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent" >
        <SurfaceView
                android:id="@+id/cameraPreview"
                android:layout_width="fill_parent"
                android:layout_height="fill_parent" />

        <TextView
            android:id="@+id/xAxisLabel"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_alignParentLeft="true"
            android:layout_alignParentTop="true"
            android:layout_marginLeft="18dp"
            android:layout_marginTop="15dp"
            android:text="@string/xAxis" />

        <TextView
            android:id="@+id/yAxisLabel"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_alignLeft="@+id/xAxisLabel"
```

```
        android:layout_below="@+id/xAxisLabel"
        android:text="@string/yAxis" />

    <TextView
        android:id="@+id/zAxisLabel"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignLeft="@+id/yAxisLabel"
        android:layout_below="@+id/yAxisLabel"
        android:text="@string/zAxis" />

    <TextView
        android:id="@+id/headingLabel"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignLeft="@+id/zAxisLabel"
        android:layout_below="@+id/zAxisLabel"
        android:layout_marginTop="19dp"
        android:text="@string/heading" />

    <TextView
        android:id="@+id/pitchLabel"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignLeft="@+id/headingLabel"
        android:layout_below="@+id/headingLabel"
        android:text="@string/pitch" />

    <TextView
        android:id="@+id/rollLabel"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignLeft="@+id/pitchLabel"
        android:layout_below="@+id/pitchLabel"
        android:text="@string/roll" />

    <TextView
        android:id="@+id/latitudeLabel"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignLeft="@+id/rollLabel"
        android:layout_below="@+id/rollLabel"
        android:layout_marginTop="34dp"
        android:text="@string/latitude" />

    <TextView
        android:id="@+id/longitudeLabel"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignLeft="@+id/latitudeLabel"
```

```
        android:layout_below="@+id/latitudeLabel"
        android:text="@string/longitude" />

    <TextView
        android:id="@+id/altitudeLabel"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignLeft="@+id/longitudeLabel"
        android:layout_below="@+id/longitudeLabel"
        android:text="@string/altitude" />

    <TextView
        android:id="@+id/xAxisValue"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignBottom="@+id/xAxisLabel"
        android:layout_marginLeft="56dp"
        android:layout_toRightOf="@+id/longitudeLabel"
        android:text="@string/empty" />

    <TextView
        android:id="@+id/yAxisValue"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignBaseline="@+id/yAxisLabel"
        android:layout_alignBottom="@+id/yAxisLabel"
        android:layout_alignLeft="@+id/xAxisValue"
        android:text="@string/empty" />

    <TextView
        android:id="@+id/zAxisValue"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_above="@+id/headingLabel"
        android:layout_alignLeft="@+id/yAxisValue"
        android:text="@string/empty" />

    <TextView
        android:id="@+id/headingValue"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignBaseline="@+id/headingLabel"
        android:layout_alignBottom="@+id/headingLabel"
        android:layout_alignLeft="@+id/zAxisValue"
        android:text="@string/empty" />

    <TextView
        android:id="@+id/pitchValue"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
```

```
            android:layout_alignBaseline="@+id/pitchLabel"
            android:layout_alignBottom="@+id/pitchLabel"
            android:layout_alignLeft="@+id/headingValue"
            android:text="@string/empty" />

        <TextView
            android:id="@+id/rollValue"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_above="@+id/latitudeLabel"
            android:layout_alignLeft="@+id/pitchValue"
            android:text="@string/empty" />

        <TextView
            android:id="@+id/latitudeValue"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_alignBottom="@+id/latitudeLabel"
            android:layout_alignLeft="@+id/rollValue"
            android:text="@string/empty" />

        <TextView
            android:id="@+id/longitudeValue"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_alignBaseline="@+id/longitudeLabel"
            android:layout_alignBottom="@+id/longitudeLabel"
            android:layout_alignLeft="@+id/latitudeValue"
            android:text="@string/empty" />

        <TextView
            android:id="@+id/altitudeValue"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_alignBaseline="@+id/altitudeLabel"
            android:layout_alignBottom="@+id/altitudeLabel"
            android:layout_alignLeft="@+id/longitudeValue"
            android:text="@string/empty" />

</RelativeLayout>
```

You can get an idea of what each `TextView` is for by looking at the IDs. Make sure that you get the layout code exactly right; otherwise, your entire layout will look like it's been pushed through a blender (the kitchen kind, not the I-can-make-cool-graphics-with-this-software kind). We will reference only `TextViews` that have "`Value`" in their IDs from the code, as the others are only labels. Those `TextViews` will be used to display the various sensor values that our app will be receiving.

TextView Variable Declarations

After the layout is in the project, we can start referencing all those TextViews from the code and updating them with the appropriate data. To be able to reference the TextViews from the code, we need to have some variables to store those references. Add the following nine variables to the top of the class (the names are self-explanatory):

Listing 3-3. *Variable Declarations*

```
TextView xAxisValue;
TextView yAxisValue;
TextView zAxisValue;
TextView headingValue;
TextView pitchValue;
TextView rollValue;
TextView altitudeValue;
TextView latitudeValue;
TextView longitudeValue;
```

Updated onCreate

After that, add the following code to the onCreate() method so that each TextView object holds a reference to the corresponding TextView in our XML.

Listing 3-4. *Supplying References to the XML TextViews*

```
xAxisValue = (TextView) findViewById(R.id.xAxisValue);
yAxisValue = (TextView) findViewById(R.id.yAxisValue);
zAxisValue = (TextView) findViewById(R.id.zAxisValue);
headingValue = (TextView) findViewById(R.id.headingValue);
pitchValue = (TextView) findViewById(R.id.pitchValue);
rollValue = (TextView) findViewById(R.id.rollValue);
altitudeValue = (TextView) findViewById(R.id.altitudeValue);
longitudeValue = (TextView) findViewById(R.id.longitudeValue);
latitudeValue = (TextView) findViewById(R.id.latitudeValue);
```

Displaying the Sensors' Data

Now that we have a reference to all the TextViews that we will be updating with our data, we should do just that. To have the ones for the accelerometer and orientation sensors updated with the correct data, modify the SensorEventListener to the following:

Listing 3-5. *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));

                    headingValue.setText(String.valueOf(headingAngle));
                    pitchValue.setText(String.valueOf(pitchAngle));
                    rollValue.setText(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));

                    xAxisValue.setText(String.valueOf(xAxis));
                    yAxisValue.setText(String.valueOf(yAxis));
                    zAxisValue.setText(String.valueOf(zAxis));
            }
    }

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

Now the sensor data is written both to the log and the `TextViews`. Because the sensor delay is set to `SENSOR_DELAY_NORMAL`, the `TextViews` will update at a moderate sort of rate. If the delay had been set to `SENSOR_DELAY_GAME`, we would have had the `TextViews` updating faster than the eye can follow. That would be very taxing on the CPU. Even now, on some of the slower devices, the app might seem to lag.

> **NOTE:** You can get around the lag by shifting the code for updating the `TextViews` into a `TimerTask` or `Handler`.

Now that the data is coming through from the orientation and accelerometer sensors, we should do the same for the GPS. This is more or less a repeat of what we did to the `SensorEventListener`, except that it is done to the `LocationListener`:

Listing 3-6. *Modified 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));

            latitudeValue.setText(String.valueOf(latitude));
            longitudeValue.setText(String.valueOf(longitude));
            altitudeValue.setText(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

            }
};
```

Once more, the data is written out to both the log and the `TextViews`. If you debug the app now, you should see one camera preview and 18 `TextViews` on the screen, of which 6 should be changing quickly, while 3 change, but more slowly. Because the GPS needs an unbroken view of the sky to work and might

take a while to get a fix on your location, the fields related to the GPS can take some time to be updated.

Updated AndroidManifest.xml

Finally, you need to change the `AndroidManifest.xml` for this project:

Listing 3-7. *Modified AndroidManifest.xml*

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.paar.ch3widgetoverlay"
    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=".ProAndroidAR3Activity"
            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>
```

These are the basics of overlaying the standard Android widgets on your camera preview. Make sure that the widget is in place and that all your IDs line up. After that, using the widgets in your app is exactly the same as in any other app. You will call the same methods, use the same functions, and do the same things. This hold true for all the widgets present in the Android framework.

Testing the App

With that, we come to the end of overlaying standard Android widgets onto your camera preview. Figures 3-2 and 3-3 show how the app should look upon completion.

Figure 3-2. *Screenshot of the app with no GPS fix*

Figure 3-3. *Screenshot of the app with a GPS fix*

Next, we will take a look at adding marker recognition to our app.

Markers

Markers are visual cues used by AR apps to know where to put overlays. You select any easily identifiable image (like a black question mark on a white background). One copy of the image is saved in your app, while another is printed out and put somewhere in the real world (or painted if you have a very steady hand). Marker recognition is an ongoing part of research in the field of artificial intelligence.

We will be using an open source Android library called AndAR to help us with the marker recognition. The details of the AndAR project can be found at `http://code.google.com/p/andar`.

Create a new project. The one on my end has the package name com.paar.ch3marker. The default activity is called `Activity.java`.

The application will have four markers that it will recognize. For each of the markers, we will supply a .patt file that AndAR can use to recognize the markers. These files describe how the markers look in a way that AndAR can understand.

You can also create and supply your own markers, if you don't like the ones provided or are feeling bored and adventurous. There are a few limitations, though:

- The marker must be square in shape.
- The borders must contrast well.
- The border must be a solid color.

The markers can be black and white or color. Figure 3-4 shows an example of a marker.

Figure 3-4. *Sample Android marker*

You can create your own markers using the online flash tool available at `http://flash.tarotaro.org/blog/2009/07/12/mgo2/`.

Activity.java

Let's start by editing `Activity.java`, which is a relatively small class.

Listing 3-8. *Modified Activity.java*

```
public class Activity extends AndARActivity {

        private ARObject someObject;
        private ARToolkit artoolkit;
        @Override
        public void onCreate(Bundle savedInstanceState) {

                super.onCreate(savedInstanceState);
                CustomRenderer renderer = new CustomRenderer();
                setNonARRenderer(renderer);
                try {
                        artoolkit = getArtoolkit();


                        someObject = new CustomObject1
                        ("test", "marker_at16.patt", 80.0, new double[]{0,0});
                        artoolkit.registerARObject(someObject);

                        someObject = new CustomObject2
                        ("test", "marker_peace16.patt", 80.0, new
double[]{0,0});
                        artoolkit.registerARObject(someObject);

                        someObject = new CustomObject3
                        ("test", "marker_rupee16.patt", 80.0, new
double[]{0,0});
                        artoolkit.registerARObject(someObject);

                        someObject = new CustomObject4
                        ("test", "marker_hand16.patt", 80.0, new double[]{0,0});
                        artoolkit.registerARObject(someObject);

                } catch (AndARException ex){
                        System.out.println("");
                }
                startPreview();
        }

        public void uncaughtException(Thread thread, Throwable ex) {
                Log.e("AndAR EXCEPTION", ex.getMessage());
                finish();
        }
}
```

In the `onCreate()` method, we first get the `savedInstanceState`. After that, we create a reference to the `CustomRenderer` class, which we will create a few pages down the line. We then set the non-AR renderer. Now comes the main

part of the class. We register all four of the markers with AndAR and their related objects. `CustomObject1-4` are classes that define what is to be augmented over each marker. Finally, the `uncaughtException()` method is used to cleanly exit the app if a fatal exception happens.

CustomObject Overlays

The custom objects are basically 3D boxes in 4 different colors. They rotate and so on depending on the view of the marker. Figure 3-5 shows one of the cubes being displayed.

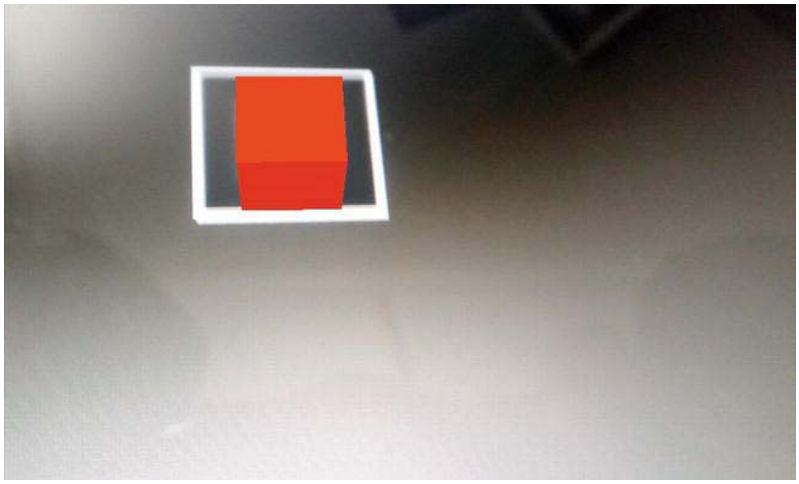

Figure 3-5. *One of the four custom object overlays*

First up is `CustomObject1.java`.

Listing 3-9. *CustomObject1*

```
public class CustomObject1 extends ARObject {


        public CustomObject1(String name, String patternName,
                        double markerWidth, double[] markerCenter) {
                super(name, patternName, markerWidth, markerCenter);
                float   mat_ambientf[]     = {0f, 1.0f, 0f, 1.0f};
                float   mat_flashf[]       = {0f, 1.0f, 0f, 1.0f};
                float   mat_diffusef[]     = {0f, 1.0f, 0f, 1.0f};
                float   mat_flash_shinyf[] = {50.0f};

                mat_ambient = GraphicsUtil.makeFloatBuffer(mat_ambientf);
```

```
                mat_flash = GraphicsUtil.makeFloatBuffer(mat_flashf);
                mat_flash_shiny =
GraphicsUtil.makeFloatBuffer(mat_flash_shinyf);
                mat_diffuse = GraphicsUtil.makeFloatBuffer(mat_diffusef);

        }
        public CustomObject1(String name, String patternName,
                        double markerWidth, double[] markerCenter, float[]
customColor) {
                super(name, patternName, markerWidth, markerCenter);
                float   mat_flash_shinyf[] = {50.0f};

                mat_ambient = GraphicsUtil.makeFloatBuffer(customColor);
                mat_flash = GraphicsUtil.makeFloatBuffer(customColor);
                mat_flash_shiny =
GraphicsUtil.makeFloatBuffer(mat_flash_shinyf);
                mat_diffuse = GraphicsUtil.makeFloatBuffer(customColor);

        }

        private SimpleBox box = new SimpleBox();
        private FloatBuffer mat_flash;
        private FloatBuffer mat_ambient;
        private FloatBuffer mat_flash_shiny;
        private FloatBuffer mat_diffuse;


        @Override
        public final void draw(GL10 gl) {
                super.draw(gl);

                gl.glMaterialfv(GL10.GL_FRONT_AND_BACK,
GL10.GL_SPECULAR,mat_flash);
                gl.glMaterialfv(GL10.GL_FRONT_AND_BACK, GL10.GL_SHININESS,
mat_flash_shiny);
                gl.glMaterialfv(GL10.GL_FRONT_AND_BACK, GL10.GL_DIFFUSE,
mat_diffuse);
                gl.glMaterialfv(GL10.GL_FRONT_AND_BACK, GL10.GL_AMBIENT,
mat_ambient);

            gl.glColor4f(0, 1.0f, 0, 1.0f);
            gl.glTranslatef( 0.0f, 0.0f, 12.5f );

            box.draw(gl);
        }
        @Override
        public void init(GL10 gl) {

        }
}
```

We start by setting the various lightings for the box and creating FloatBuffers out of them in the constructors. We then get a simple box directly from AndAR, so that we are spared the trouble of making it. In the draw() method, we draw everything. In this case, everything done in the draw() method will be done directly on the marker.

The other three CustomObject classes are exactly the same as CustomObject1, except we change the colors a bit. Following are the changes you need to make for CustomObject2.

Listing 3-10. *CustomObject2*

```
        public CustomObject2(String name, String patternName,
                        double markerWidth, double[] markerCenter) {
                super(name, patternName, markerWidth, markerCenter);
                float   mat_ambientf[]     = {1.0f, 0f, 0f, 1.0f};
                float   mat_flashf[]       = {1.0f, 0f, 0f, 1.0f};
                float   mat_diffusef[]     = {1.0f, 0f, 0f, 1.0f};
                float   mat_flash_shinyf[] = {50.0f};

                mat_ambient = GraphicsUtil.makeFloatBuffer(mat_ambientf);
                mat_flash = GraphicsUtil.makeFloatBuffer(mat_flashf);
                mat_flash_shiny =
GraphicsUtil.makeFloatBuffer(mat_flash_shinyf);
                mat_diffuse = GraphicsUtil.makeFloatBuffer(mat_diffusef);

        }

//Same code everywhere else, except the draw() method

        @Override
        public final void draw(GL10 gl) {
                super.draw(gl);

                gl.glMaterialfv(GL10.GL_FRONT_AND_BACK,
GL10.GL_SPECULAR,mat_flash);
                gl.glMaterialfv(GL10.GL_FRONT_AND_BACK, GL10.GL_SHININESS,
mat_flash_shiny);
                gl.glMaterialfv(GL10.GL_FRONT_AND_BACK, GL10.GL_DIFFUSE,
mat_diffuse);
                gl.glMaterialfv(GL10.GL_FRONT_AND_BACK, GL10.GL_AMBIENT,
mat_ambient);

            gl.glColor4f(1.0f, 0, 0, 1.0f);
            gl.glTranslatef( 0.0f, 0.0f, 12.5f );

            box.draw(gl);
        }
```

Following are the changes for CustomObject3.

Listing 3-11. *CustomObject3*

```
        public CustomObject3(String name, String patternName,
                        double markerWidth, double[] markerCenter) {
                super(name, patternName, markerWidth, markerCenter);
                float   mat_ambientf[]     = {0f, 0f, 1.0f, 1.0f};
                float   mat_flashf[]       = {0f, 0f, 1.0f, 1.0f};
                float   mat_diffusef[]     = {0f, 0f, 1.0f, 1.0f};
                float   mat_flash_shinyf[] = {50.0f};

                mat_ambient = GraphicsUtil.makeFloatBuffer(mat_ambientf);
                mat_flash = GraphicsUtil.makeFloatBuffer(mat_flashf);
                mat_flash_shiny =
GraphicsUtil.makeFloatBuffer(mat_flash_shinyf);
                mat_diffuse = GraphicsUtil.makeFloatBuffer(mat_diffusef);

        }

//Same code everywhere else, except the draw() method

        @Override
        public final void draw(GL10 gl) {
                super.draw(gl);

                gl.glMaterialfv(GL10.GL_FRONT_AND_BACK,
GL10.GL_SPECULAR,mat_flash);
                gl.glMaterialfv(GL10.GL_FRONT_AND_BACK, GL10.GL_SHININESS,
mat_flash_shiny);
                gl.glMaterialfv(GL10.GL_FRONT_AND_BACK, GL10.GL_DIFFUSE,
mat_diffuse);
                gl.glMaterialfv(GL10.GL_FRONT_AND_BACK, GL10.GL_AMBIENT,
mat_ambient);

            gl.glColor4f(0f, 0, 1.0, 1.0f);
            gl.glTranslatef( 0.0f, 0.0f, 12.5f );

            box.draw(gl);
        }
```

And finally, the changes for CustomObject4 follow.

Listing 3-12. *CustomObject4*

```
        public CustomObject4(String name, String patternName,
                        double markerWidth, double[] markerCenter) {
                super(name, patternName, markerWidth, markerCenter);
                float   mat_ambientf[]     = {1.0f, 0f, 1.0f, 1.0f};
                float   mat_flashf[]       = {1.0f, 0f, 1.0f, 1.0f};
                float   mat_diffusef[]     = {1.0f, 0f, 1.0f, 1.0f};
                float   mat_flash_shinyf[] = {50.0f};

                mat_ambient = GraphicsUtil.makeFloatBuffer(mat_ambientf);
                mat_flash = GraphicsUtil.makeFloatBuffer(mat_flashf);
                mat_flash_shiny =
GraphicsUtil.makeFloatBuffer(mat_flash_shinyf);
                mat_diffuse = GraphicsUtil.makeFloatBuffer(mat_diffusef);

        }

//Same code everywhere else, except the draw() method

        @Override
        public final void draw(GL10 gl) {
                super.draw(gl);

                gl.glMaterialfv(GL10.GL_FRONT_AND_BACK,
GL10.GL_SPECULAR,mat_flash);
                gl.glMaterialfv(GL10.GL_FRONT_AND_BACK, GL10.GL_SHININESS,
mat_flash_shiny);
                gl.glMaterialfv(GL10.GL_FRONT_AND_BACK, GL10.GL_DIFFUSE,
mat_diffuse);
                gl.glMaterialfv(GL10.GL_FRONT_AND_BACK, GL10.GL_AMBIENT,
mat_ambient);

            gl.glColor4f(1.0f, 0, 1.0, 1.0f);
            gl.glTranslatef( 0.0f, 0.0f, 12.5f );

            box.draw(gl);
        }
```

CustomRenderer

Now we have only `CustomRenderer.java` to deal with. This class allows us to do any non-AR stuff as well as set up the OpenGL environment.

Listing 3-13. *CustomRenderer*

```
public class CustomRenderer implements OpenGLRenderer {
```

```
        private float[] ambientlight1 = {.3f, .3f, .3f, 1f};
        private float[] diffuselight1 = {.7f, .7f, .7f, 1f};
        private float[] specularlight1 = {0.6f, 0.6f, 0.6f, 1f};
        private float[] lightposition1 = {20.0f,-40.0f,100.0f,1f};

        private FloatBuffer lightPositionBuffer1 =
GraphicsUtil.makeFloatBuffer(lightposition1);
        private FloatBuffer specularLightBuffer1 =
GraphicsUtil.makeFloatBuffer(specularlight1);
        private FloatBuffer diffuseLightBuffer1 =
GraphicsUtil.makeFloatBuffer(diffuselight1);
        private FloatBuffer ambientLightBuffer1 =
GraphicsUtil.makeFloatBuffer(ambientlight1);


        public final void draw(GL10 gl) {
        }


        public final void setupEnv(GL10 gl) {
                gl.glEnable(GL10.GL_LIGHTING);
                gl.glLightfv(GL10.GL_LIGHT1, GL10.GL_AMBIENT,
ambientLightBuffer1);
                gl.glLightfv(GL10.GL_LIGHT1, GL10.GL_DIFFUSE,
diffuseLightBuffer1);
                gl.glLightfv(GL10.GL_LIGHT1, GL10.GL_SPECULAR,
specularLightBuffer1);
                gl.glLightfv(GL10.GL_LIGHT1, GL10.GL_POSITION,
lightPositionBuffer1);
                gl.glEnable(GL10.GL_LIGHT1);
                gl.glDisableClientState(GL10.GL_TEXTURE_COORD_ARRAY);
                gl.glDisable(GL10.GL_TEXTURE_2D);
                initGL(gl);
        }

        public final void initGL(GL10 gl) {
                gl.glDisable(GL10.GL_COLOR_MATERIAL);
                gl.glEnable(GL10.GL_CULL_FACE);
                gl.glShadeModel(GL10.GL_SMOOTH);
                gl.glDisable(GL10.GL_COLOR_MATERIAL);
                gl.glEnable(GL10.GL_LIGHTING);
                gl.glEnable(GL10.GL_CULL_FACE);
                gl.glEnable(GL10.GL_DEPTH_TEST);
                gl.glEnable(GL10.GL_NORMALIZE);
        }
}
```

In the variable declarations, we specify the different types of lighting and create `FloatBuffers` from them. The `setupEnv()` is called before we display any of the

boxes. It sets up the lighting and other OpenGL-specific stuff. The `initGL()` method is called once when the Surface is created.

AndroidManifest

Finally, the `AndroidManifest.xml` needs to be updated.

Listing 3-14. *Updated AndroidManifest.xml*

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.paar.ch3marker"
    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=".Activity"
            android:clearTaskOnLaunch="true"
            android:screenOrientation="landscape"
            android:noHistory="true">
            <intent-filter >
                <action android:name="android.intent.action.MAIN" />

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

This brings us to the end of the coding for this app. If you have not downloaded the source code for this chapter yet, please do so and take the .patt files and put them in your projects /assets directory. Along with the source, you will find a folder called “Markers,” which contains the markers used in this app and further on in the book. You can print them for your use.

Summary

In this chapter, we learned how to overlay standard Android widgets in our app and how to use markers to make our augmented reality apps more interactive. about the chapter also discussed AndAR, an opensource AR toolkit available for Android that allows us to implement a lot of AR features painlessly and quickly.

The next chapter discusses Artificial Horizon, a feature of AR that is central to any military or navigational app.