Chapter 9

An Augmented Reality Browser

Welcome to Chapter 9. This is the last chapter in this book, which has covered the different aspects of augmented reality (AR) from making a basic app, using markers, overlaying widgets, and making navigational apps. This final chapter discusses the example app of a real-world AR browser. This browser is something along the lines of the extremely popular Wikitude and Layar apps, but not quite as extensive. Wikitude and Layar are very powerful tools that were developed over a long period of time and provide many, many features. Our AR browser will be relatively humble, but still very, very powerful and useful:

- It will have a live camera preview
- Twitter posts and topics of Wikipedia articles that are located nearby will be displayed over this preview
- There will be a small radar visible that allows the user to see whether any other overlays are available outside their field of view
- Overlays will be moved in and out of the view as the user moves and rotates
- The user can set the radius of data collection from 0m to 100,000m (100 km)

Figure 9-1 shows the app running, with markers from both data sources visible.

Figure 9-1. *A screenshot of the app when it is running.*

To accomplish all this, we will write our own mini AR engine and use two freely available resources to get the Wikipedia and Twitter data. Compared with Chapter 8, the code isn't very long, but some of it is new, especially the moving overlays part. Without further ado, let's get coding.

The XML

The XML in this app consists of only strings.xml and the menu's XML. We'll quickly type those up and then move onto the Java code.

strings.xml

Listing 9-1. *strings.xml*

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">Pro Android AR Chapter 9</string>
</resources>
```

The string `app_name` simply stores the name of our app. This name is displayed under the icon in the launcher.

menu.xml

Now let's take a look at menu.xml.

Listing 9-2. *menu.xml*

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/showRadar"
          android:title="Hide Radar">
    </item>
    <item android:id="@+id/showZoomBar"
          android:title="Hide Zoom Bar">
    </item>
    <item android:id="@+id/exit"
          android:title="Exit">
    </item>
</menu>
```

The first item is the toggle to show and hide the radar that will be used to display the icons for objects outside the user's field of view. The second item is the toggle to show and hide the SeekBar widget that allows the user to adjust the radius of the tweets and Wikipedia information.

With that little bit of XML out of the way, we can move on to the Java code of our app.

The Java Code

In this app, we'll take a look at the Java code in a format in which different classes are grouped by functions. So we'll look at all the data-parsing classes in sequence and so on.

The Activities and AugmentedView

SensorsActivity

Let's start with the basic parts of our app. We have one `SensorsActivity`, which extends the standard android `Activity`. `SensorsActivity` has no user interface. `AugmentedActivity` then extends this `SensorsActivity`, which is extended by `MainActivity`, which is the `Activity` that will finally be displayed to the user. So let's start by taking a look at `SensorsActivity`.

Listing 9-3. *SensorsActivity.java Global Variables*

```
public class SensorsActivity extends Activity implements SensorEventListener,
LocationListener {
    private static final String TAG = "SensorsActivity";
    private static final AtomicBoolean computing = new AtomicBoolean(false);
```

```
private static final int MIN_TIME = 30*1000;
private static final int MIN_DISTANCE = 10;

private static final float temp[] = new float[9];
private static final float rotation[] = new float[9];
private static final float grav[] = new float[3];
private static final float mag[] = new float[3];

private static final Matrix worldCoord = new Matrix();
private static final Matrix magneticCompensatedCoord = new Matrix();
private static final Matrix xAxisRotation = new Matrix();
private static final Matrix magneticNorthCompensation = new Matrix();

private static GeomagneticField gmf = null;
private static float smooth[] = new float[3];
private static SensorManager sensorMgr = null;
private static List<Sensor> sensors = null;
private static Sensor sensorGrav = null;
private static Sensor sensorMag = null;
private static LocationManager locationMgr = null;
```

The first variable is simply a `TAG` constant with our class' name in it. The second one, `computing`, which is like a flag, is used to check if a task is currently in progress. `MIN_TIME` and `MIN_DISTANCE` specify the minimum time and distance between location updates. Of the four floats up there, the first is a temporary array used while rotating, the second stores the final rotated matrix, `grav` stores the gravity numbers, and `mag` stores the magnetic field numbers. In the four matrices after that, `worldCoord` stores the location of the device on the world, `magneticCompensatedCoord` and `magneticNorthCompensation` are used when compensating for the difference in between the geographical north pole and the magnetic north pole, and `xAxisRotation` is used to store the matrix after it has been rotated by 90 degrees along the X-axis. After that, `gmf` is used to store an instance of the `GeomagneticField` later on in the class. The `smooth` array is used when using a low-pass Filter on the values from `grav` and `mag`. `sensorMgr` is a `SensorManager` object; `sensors` is a list of sensors. `sensorGrav` and `sensorMag` will store the default gravitational (accelerometer) and magnetic (compass) sensors on the device, just in case the device has more than one sensor. `locationMgr` is an instance of the `LocationManager`.

Now let's take a look at our methods. In this particular `Activity`, we do not need to do anything in `onCreate()`, so we just implement a basic version of it so that classes that extend this one can use it. Our main work is done in the `onStart()` method:

Listing 9-4. *SensorsActivity.java onCreate() and onStart()*

```
@Override
   public void onCreate(Bundle savedInstanceState) {
       super.onCreate(savedInstanceState);
   }

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

       double angleX = Math.toRadians(-90);
       double angleY = Math.toRadians(-90);

       xAxisRotation.set( 1f,
               0f,
               0f,
               0f,
               (float) Math.cos(angleX),
               (float) -Math.sin(angleX),
               0f,
               (float) Math.sin(angleX),
               (float) Math.cos(angleX));

       try {
           sensorMgr = (SensorManager) getSystemService(SENSOR_SERVICE);

           sensors = sensorMgr.getSensorList(Sensor.TYPE_ACCELEROMETER);

           if (sensors.size() > 0) sensorGrav = sensors.get(0);

           sensors = sensorMgr.getSensorList(Sensor.TYPE_MAGNETIC_FIELD);

           if (sensors.size() > 0) sensorMag = sensors.get(0);

           sensorMgr.registerListener(this, sensorGrav,
SensorManager.SENSOR_DELAY_NORMAL);
           sensorMgr.registerListener(this, sensorMag,
SensorManager.SENSOR_DELAY_NORMAL);

           locationMgr = (LocationManager)
getSystemService(Context.LOCATION_SERVICE);
           locationMgr.requestLocationUpdates(LocationManager.GPS_PROVIDER,
MIN_TIME, MIN_DISTANCE, this);

           try {

               try {
                   Location
gps=locationMgr.getLastKnownLocation(LocationManager.GPS_PROVIDER);
```

```
                    Location
network=locationMgr.getLastKnownLocation(LocationManager.NETWORK_PROVIDER);
                    if(gps!=null)
                    {
                        onLocationChanged(gps);
                    }
                    else if (network!=null)
                    {
                        onLocationChanged(network);
                    }
                    else
                    {
                        onLocationChanged(ARData.hardFix);
                    }
                } catch (Exception ex2) {
                    onLocationChanged(ARData.hardFix);
                }

                gmf = new GeomagneticField((float)
ARData.getCurrentLocation().getLatitude(),
                                           (float)
ARData.getCurrentLocation().getLongitude(),
                                           (float)
ARData.getCurrentLocation().getAltitude(),
                                           System.currentTimeMillis());
                angleY = Math.toRadians(-gmf.getDeclination());

                synchronized (magneticNorthCompensation) {

                    magneticNorthCompensation.toIdentity();

                    magneticNorthCompensation.set( (float) Math.cos(angleY),
                                                   0f,
                                                   (float) Math.sin(angleY),
                                                   0f,
                                                   1f,
                                                   0f,
                                                   (float) -Math.sin(angleY),
                                                   0f,
                                                   (float) Math.cos(angleY));

                    magneticNorthCompensation.prod(xAxisRotation);
                }
            } catch (Exception ex) {
                ex.printStackTrace();
            }
        } catch (Exception ex1) {
            try {
                if (sensorMgr != null) {
                    sensorMgr.unregisterListener(this, sensorGrav);
```

```
                sensorMgr.unregisterListener(this, sensorMag);
                sensorMgr = null;
            }
            if (locationMgr != null) {
                locationMgr.removeUpdates(this);
                locationMgr = null;
            }
        } catch (Exception ex2) {
            ex2.printStackTrace();
        }
    }
}
```

As mentioned before the code, the `onCreate()` method is simply a default implementation. The `onStart()` method, on the other hand, has a good amount of sensor and location related code in it. We begin by setting the value of the `xAxisRotation` Matrix. We then set the values for `sensorMag` and `sensorGrav` and then register the two sensors. We then assign a value to `gmf` and reassign the value of `angleY` to the negative declination , which is the difference between true north (North Pole) and magnetic north (currently located somewhere off the coast of Greenland, moving toward Siberia at roughly 40 miles/year) of `gmf` in radians. After this reassignment, we have some code in a synchronized block. This code is used to first set the value of `magneticNorthCompensation` and then multiply it with `xAxisRotation`. After this, we have a `catch` block, followed by another `catch` block with a `try` block inside it. This `try` block's code simply attempts to unregister the sensors and location listeners.

Next in this file is our `onStop()` method, which is the same as the `onResume()` and `onStop()` methods used previously in the book. We simply use it to let go of the sensors and GPS to save the user's battery life, and stop collecting data when it is not required.

Listing 9-5. *SensorsActivity.java onStop()*

```
@Override
   protected void onStop() {
       super.onStop();

       try {
           try {
               sensorMgr.unregisterListener(this, sensorGrav);
               sensorMgr.unregisterListener(this, sensorMag);
           } catch (Exception ex) {
               ex.printStackTrace();
           }
           sensorMgr = null;

           try {
```

```
                locationMgr.removeUpdates(this);
            } catch (Exception ex) {
                ex.printStackTrace();
            }
            locationMgr = null;
        } catch (Exception ex) {
                ex.printStackTrace();
        }
    }
```

After the `onStop()` method, we have the methods for getting the data from the three sensors. If you look at the class declaration given previously, you'll notice that this time we implement `SensorEventListener` and `LocationListener` for the entire class instead of having small code blocks for them, as we did in our previous apps. We do this so that any class that extends this class can easily override the sensor-related methods.

Listing 9-6. *SensorsActivity.java Listening to the Sensors*

```
public void onSensorChanged(SensorEvent evt) {
        if (!computing.compareAndSet(false, true)) return;

        if (evt.sensor.getType() == Sensor.TYPE_ACCELEROMETER) {
            smooth = LowPassFilter.filter(0.5f, 1.0f, evt.values, grav);
            grav[0] = smooth[0];
            grav[1] = smooth[1];
            grav[2] = smooth[2];
        } else if (evt.sensor.getType() == Sensor.TYPE_MAGNETIC_FIELD) {
            smooth = LowPassFilter.filter(2.0f, 4.0f, evt.values, mag);
            mag[0] = smooth[0];
            mag[1] = smooth[1];
            mag[2] = smooth[2];
        }

        SensorManager.getRotationMatrix(temp, null, grav, mag);

        SensorManager.remapCoordinateSystem(temp, SensorManager.AXIS_Y,
SensorManager.AXIS_MINUS_X, rotation);

        worldCoord.set(rotation[0], rotation[1], rotation[2], rotation[3],
rotation[4], rotation[5], rotation[6], rotation[7], rotation[8]);

        magneticCompensatedCoord.toIdentity();

        synchronized (magneticNorthCompensation) {
            magneticCompensatedCoord.prod(magneticNorthCompensation);
        }

        magneticCompensatedCoord.prod(worldCoord);
```

```
        magneticCompensatedCoord.invert();

        ARData.setRotationMatrix(magneticCompensatedCoord);

        computing.set(false);
    }

    public void onProviderDisabled(String provider) {
        //Not Used
    }

    public void onProviderEnabled(String provider) {
        //Not Used
    }

    public void onStatusChanged(String provider, int status, Bundle extras) {
        //Not Used
    }

    public void onLocationChanged(Location location) {
        ARData.setCurrentLocation(location);
        gmf = new GeomagneticField((float)
ARData.getCurrentLocation().getLatitude(),
                (float) ARData.getCurrentLocation().getLongitude(),
                (float) ARData.getCurrentLocation().getAltitude(),
                System.currentTimeMillis());

        double angleY = Math.toRadians(-gmf.getDeclination());

        synchronized (magneticNorthCompensation) {
            magneticNorthCompensation.toIdentity();

            magneticNorthCompensation.set((float) Math.cos(angleY),
                                          0f,
                                          (float) Math.sin(angleY),
                                          0f,
                                          1f,
                                          0f,
                                          (float) -Math.sin(angleY),
                                          0f,
                                          (float) Math.cos(angleY));

            magneticNorthCompensation.prod(xAxisRotation);
        }
    }

    public void onAccuracyChanged(Sensor sensor, int accuracy) {
                if (sensor==null) throw new NullPointerException();
```

```
        if(sensor.getType() == Sensor.TYPE_MAGNETIC_FIELD &&
accuracy==SensorManager.SENSOR_STATUS_UNRELIABLE) {
            Log.e(TAG, "Compass data unreliable");
        }
    }
}
```

Six methods are present in this code. These are `onSensorChanged()`, `onProviderDisabled()`, `onProviderEnabled()`, `onStatusChanged`, `onLocationChanged()`, and `onAccuracyChanged()`. `onProviderDisabled()`, `onProviderEnabled()` and `onStatusChanged()` are not used, but are still there because they must be implemented.

Now let's take a look at the three that are used. In `onSensorChanged()` we first get the sensor values from the compass and accelerometer and pass them through a low-pass filter before storing them. The low pass filter is discussed and explained in detail later on in this chapter when we write the code for it. After storing the values, we find out the real world coordinates and store them in the temp array. Immediately after that we remap the coordinates to work with the Android device in landscape mode and store it in the rotation array. We then convert the rotation array to a matrix by transferring the data into the `worldCoord` matrix. We then multiply `magneticCompensatedCoord` with `magneticNorthCompensation` and then multiply `magneticCompensatedCoord` with `worldCoord`. `magneticCompensatedCoord` is then inverted, and set as the rotation matrix for `ARData`. This rotation matrix will be used to convert the latitude and longitude of our tweets and Wikipedia articles to X and Y coordinates for our display.

In `onLocationChanged()`, we first update the location in `ARData`, recalculate the `gmf` with the new data, and then execute the same code as we did in `onStart()`.

In `onAccuracyChanged()`, we check to see whether the data is `null` first. If it is, a `NullPointerException` is thrown. If the data isn't `null`, and the compass seems to have become unreliable, we add an error message to the `LogCat` saying so.

AugmentedView

Before we move on to `AugmentedActivity`, we need to create `AugmentedView`, which is our own custom extension of the `View` class found in the Android framework. It is designed to draw the Radar, the zoom bar controlling the radius of the data and the markers that show the data over our camera preview. Let's start with the class and global variable declarations.

Listing 9-7. *Declaring AugmentedView and its Variables*

```
public class AugmentedView extends View {
    private static final AtomicBoolean drawing = new AtomicBoolean(false);
    private static final Radar radar = new Radar();
    private static final float[] locationArray = new float[3];
    private static final List<Marker> cache = new ArrayList<Marker>();
    private static final TreeSet<Marker> updated = new TreeSet<Marker>();
    private static final int COLLISION_ADJUSTMENT = 100;
```

The `AtomicBoolean` drawing is a flag to check whether the process of drawing is currently going on. `radar` is an instance of the `Radar` class, which we will write later in the book after we have finished with all the Activities. `locationArray` is used to store the locations of the markers we work on later on. `cache` is used as, well, a temporary cache when drawing. `updated` is used when we have updated the data we get from our data sources for the Wikipedia articles and tweets. `COLLISION_ADJUSTMENT` is used to adjust the locations of the markers onscreen, so that they do not overlap each other.

Now let's take a look at its constructor and the `onDraw()` method.

Listing 9-8. *The onDraw() Method and the Constructor for AugmentedView*

```
public AugmentedView(Context context) {
        super(context);
    }

    @Override
    protected void onDraw(Canvas canvas) {
        if (canvas==null) return;

        if (drawing.compareAndSet(false, true)) {
                List<Marker> collection = ARData.getMarkers();

            cache.clear();
            for (Marker m : collection) {
                m.update(canvas, 0, 0);
                if (m.isOnRadar()) cache.add(m);
                }
            collection = cache;

                if (AugmentedActivity.useCollisionDetection)
adjustForCollisions(canvas,collection);

                        ListIterator<Marker> iter =
collection.listIterator(collection.size());
                        while (iter.hasPrevious()) {
                        Marker marker = iter.previous();
                        marker.draw(canvas);
```

```
                }
                if (AugmentedActivity.showRadar) radar.draw(canvas);
                drawing.set(false);
            }
        }
```

The constructor for this class merely ties it to View via `super()`. In the `onDraw()` method, we first add all the markers that are on the radar to the cache variable and then duplicate it into collection. Then the markers are adjusted for collision (see the next code listing for details), and finally all the markers are drawn, and the radar is updated.

Now let's take a look at the code for adjusting the markers for collision:

Listing 9-9. *Adjusting for Collisions*

```
private static void adjustForCollisions(Canvas canvas, List<Marker> collection)
{
        updated.clear();
        for (Marker marker1 : collection) {
            if (updated.contains(marker1) || !marker1.isInView()) continue;

            int collisions = 1;
            for (Marker marker2 : collection) {
                if (marker1.equals(marker2) || updated.contains(marker2) ||
!marker2.isInView()) continue;

                if (marker1.isMarkerOnMarker(marker2)) {
                    marker2.getLocation().get(locationArray);
                    float y = locationArray[1];
                    float h = collisions*COLLISION_ADJUSTMENT;
                    locationArray[1] = y+h;
                    marker2.getLocation().set(locationArray);
                    marker2.update(canvas, 0, 0);
                    collisions++;
                    updated.add(marker2);
                }
            }
            updated.add(marker1);
        }
}
}      //Closes class
```

We use this code to check whether one or more markers overlap another marker and then adjust their location data so that when they are drawn, they do not overlap. We use methods from the `marker` class (written later on in this chapter) to check whether the markers are overlapping and then adjust their location in our `locationArray` appropriately.

AugmentedActivity

Now that we have written up AugmentedView, we can get to work on AugmentedActivity. We had to extend the view class first because we will be using AugmentedView in AugmentedActivity. Let's start with class and global variables.

Listing 9-10. *Declaring AugmentedActivity and its Global Variables*

```
public class AugmentedActivity extends SensorsActivity implements
OnTouchListener {
    private static final String TAG = "AugmentedActivity";
    private static final DecimalFormat FORMAT = new DecimalFormat("#.##");
    private static final int ZOOMBAR_BACKGROUND_COLOR =
Color.argb(125,55,55,55);
    private static final String END_TEXT =
FORMAT.format(AugmentedActivity.MAX_ZOOM)+" km";
    private static final int END_TEXT_COLOR = Color.WHITE;

    protected static WakeLock wakeLock = null;
    protected static CameraSurface camScreen = null;
    protected static VerticalSeekBar myZoomBar = null;
    protected static TextView endLabel = null;
    protected static LinearLayout zoomLayout = null;
    protected static AugmentedView augmentedView = null;

    public static final float MAX_ZOOM = 100; //in KM
    public static final float ONE_PERCENT = MAX_ZOOM/100f;
    public static final float TEN_PERCENT = 10f*ONE_PERCENT;
    public static final float TWENTY_PERCENT = 2f*TEN_PERCENT;
    public static final float EIGHTY_PERCENTY = 4f*TWENTY_PERCENT;

    public static boolean useCollisionDetection = true;
    public static boolean showRadar = true;
    public static boolean showZoomBar = true;
```

TAG is used as a string constant when outputting to the LogCat. FORMAT is used to format the output when displaying the current radius on the radar. ZOOMBAR_BACKGROUND_COLOR is an ARGB_8888 definition for the background color of the slider we use to allow the user to alter the radius. END_TEXT is the formatted piece of text we need to display on the radar. END_TEXT_COLOR is the color of the END_TEXT. wakeLock, camScreen, myZoomBar, endLabel, zoomLayout, and augmentedView are objects of the classes we need. They are all currently given a null value and will be initialized later on in the chapter. MAX_ZOOM is our radius limit in kilometers. The four floats that follow are various percentages of this maximum radius limit. useCollisionDetection is a flag that allows us to enable or disable collisions detection for the markers. showRadar is a flag the

toggles the visibility of the radar. showZoomBar does the same toggle, except it does so for the seekbar that controls the radius.

Now let's take a look at the onCreate() method of this Activity:

Listing 9-11. *AugmentedActivity onCreate()*

```
 @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        camScreen = new CameraSurface(this);
        setContentView(camScreen);

        augmentedView = new AugmentedView(this);
        augmentedView.setOnTouchListener(this);
        LayoutParams augLayout = new LayoutParams(  LayoutParams.WRAP_CONTENT,
LayoutParams.WRAP_CONTENT);
        addContentView(augmentedView,augLayout);

        zoomLayout = new LinearLayout(this);

zoomLayout.setVisibility((showZoomBar)?LinearLayout.VISIBLE:LinearLayout.GONE);
        zoomLayout.setOrientation(LinearLayout.VERTICAL);
        zoomLayout.setPadding(5, 5, 5, 5);
        zoomLayout.setBackgroundColor(ZOOMBAR_BACKGROUND_COLOR);

        endLabel = new TextView(this);
        endLabel.setText(END_TEXT);
        endLabel.setTextColor(END_TEXT_COLOR);
        LinearLayout.LayoutParams zoomTextParams =  new
LinearLayout.LayoutParams(LayoutParams.WRAP_CONTENT,LayoutParams.WRAP_CONTENT);
        zoomLayout.addView(endLabel, zoomTextParams);

        myZoomBar = new VerticalSeekBar(this);
        myZoomBar.setMax(100);
        myZoomBar.setProgress(50);
        myZoomBar.setOnSeekBarChangeListener(myZoomBarOnSeekBarChangeListener);
        LinearLayout.LayoutParams zoomBarParams =  new
LinearLayout.LayoutParams(LayoutParams.WRAP_CONTENT, LayoutParams.FILL_PARENT);
        zoomBarParams.gravity = Gravity.CENTER_HORIZONTAL;
        zoomLayout.addView(myZoomBar, zoomBarParams);

        FrameLayout.LayoutParams frameLayoutParams = new
FrameLayout.LayoutParams(  LayoutParams.WRAP_CONTENT, LayoutParams.FILL_PARENT,
Gravity.RIGHT);
        addContentView(zoomLayout,frameLayoutParams);

        updateDataOnZoom();
```

```
        PowerManager pm = (PowerManager)
getSystemService(Context.POWER_SERVICE);
wakeLock = pm.newWakeLock(PowerManager.SCREEN_DIM_WAKE_LOCK, "DimScreen");
 }
```

We begin by assigning and instance of `CameraSurface` to `camScreen`. We will be writing `CameraSurface` a little later in the chapter, but basically it deals with the setup of the Camera's surface view, as we have done numerous times in the previous chapters. We then set the basic content view to this `CameraSurface`. After this, we create an instance of `AugmentedView`, set its layout parameters to `WRAP_CONTENT`, and add it to the screen. We then add the base layout for the SeekBar and the `END_TEXT` to the screen. We then add the SeekBar to the screen and call a method to update the data. Finally, we use the `PowerManager` to acquire a `WakeLock` to keep the screen on, but dim it if not in use.

We then have the `onPause()` and `onResume()` methods, in which we simply release and reacquire the `WakeLock`:

Listing 9-12. *onPause() and onResume()*

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

 wakeLock.acquire();
}

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

 wakeLock.release();
}
```

Now we have our `onSensorChanged()` method:

Listing 9-13. *onSensorChanged()*

```
@Override
 public void onSensorChanged(SensorEvent evt) {
        super.onSensorChanged(evt);

        if (evt.sensor.getType() == Sensor.TYPE_ACCELEROMETER ||
             evt.sensor.getType() == Sensor.TYPE_MAGNETIC_FIELD)
         {
               augmentedView.postInvalidate();
         }
}
```

We use the method to listen for changes on the compass and accelerometer sensors. If any of those sensors change, we invalidate the `augmentedView` by calling `postInvalidate()`. It automatically calls `invalidate()`, which calls the `onDraw()` of the view.

We then have the methods that handle the changes for the SeekBar:

Listing 9-14. *Handling the SeekBar*

```
private OnSeekBarChangeListener myZoomBarOnSeekBarChangeListener = new
OnSeekBarChangeListener() {
        public void onProgressChanged(SeekBar seekBar, int progress, boolean
fromUser) {
            updateDataOnZoom();
            camScreen.invalidate();
        }

        public void onStartTrackingTouch(SeekBar seekBar) {
            //Not used
        }

        public void onStopTrackingTouch(SeekBar seekBar) {
            updateDataOnZoom();
            camScreen.invalidate();
        }
    };
```

In the methods that are called while the SeekBar is being altered (`onProgressChanged()`), and after it has stopped being altered (`onStopTrackingTouch()`), we update our data by calling `updateDataOnZoom()` and then invalidate the camera preview.

Now we have a method to calculate the zoom level of our app. We call it *zoom level*, but it is actually the radius in which we are displaying data. It's just that zoom level is easier to remember and say than *radius level*.

Listing 9-15. *Calculating the Zoom Level*

```
private static float calcZoomLevel(){
        int myZoomLevel = myZoomBar.getProgress();
        float out = 0;

        float percent = 0;
        if (myZoomLevel <= 25) {
            percent = myZoomLevel/25f;
            out = ONE_PERCENT*percent;
        } else if (myZoomLevel > 25 && myZoomLevel <= 50) {
            percent = (myZoomLevel-25f)/25f;
            out = ONE_PERCENT+(TEN_PERCENT*percent);
```

```
        } else if (myZoomLevel > 50 && myZoomLevel <= 75) {
            percent = (myZoomLevel-50f)/25f;
            out = TEN_PERCENT+(TWENTY_PERCENT*percent);
        } else {
            percent = (myZoomLevel-75f)/25f;
            out = TWENTY_PERCENT+(EIGHTY_PERCENTY*percent);
        }
        return out;
    }
```

We start by getting the progress on the SeekBar as the zoom level. We then create the float `out` to store the final result and float `percent` to store the percentage. We then have some simple math to determine the radius being used. We use these kinds of calculations because it allows the user to set the radius even in meters. The higher up on the bar you go, the less accurate the radius setting becomes. Finally, we return `out` as the current zoom level.

We now have the methods for dealing with touches and updating the data.

Listing 9-16. *Updating the Data and Handling Touch*

```
protected void updateDataOnZoom() {
        float zoomLevel = calcZoomLevel();
        ARData.setRadius(zoomLevel);
        ARData.setZoomLevel(FORMAT.format(zoomLevel));
        ARData.setZoomProgress(myZoomBar.getProgress());
    }

        public boolean onTouch(View view, MotionEvent me) {
            for (Marker marker : ARData.getMarkers()) {
                if (marker.handleClick(me.getX(), me.getY())) {
                    if (me.getAction() == MotionEvent.ACTION_UP)
markerTouched(marker);
                    return true;
                }
            }
  return               super.onTouchEvent(me);
 };

        protected void markerTouched(Marker marker) {
                Log.w(TAG, "markerTouched() not implemented.");
 }
}
```

In `updateDataOnZoom()`, we get the zoom level, set the radius to the new zoom level, and update the text for the zoom level and the seek bar's progress, all in `ARData`. In `onTouch()`, we check if a marker has been touched, and the call `markerTouched()` from there. After that, `markerTouched()` puts a message out to the LogCat saying that we currently do nothing in `markerTouched()`.

This brings us to the end of AugmentedActivity. Now we need to write our final Activity class: MainActivity.

MainActivity

Once again, let's start with the class and global variable declarations:

Listing 9-17. *Declaring the Class and Global Variables*

```
public class MainActivity extends AugmentedActivity {
    private static final String TAG = "MainActivity";
    private static final String locale = "en";
    private static final BlockingQueue<Runnable> queue = new
ArrayBlockingQueue<Runnable>(1);
    private static final ThreadPoolExecutor exeService = new
ThreadPoolExecutor(1, 1, 20, TimeUnit.SECONDS, queue);
    private static final Map<String,NetworkDataSource> sources = new
ConcurrentHashMap<String,NetworkDataSource>();
```

TAG serves the same purpose as in the previous classes that we wrote. The locale string stores the locale in the two-letter code as English. You can also use Locale.getDefault().getLanguage() to get the locale, but it is best to leave it as "en" because we use it to get the Twitter and Wikipedia data nearby, and our data sources may not support all languages. To simplify our threading, we have the queue variable that is an instance of BlockingQueue. exeService is a ThreadPoolExecutor with queue as its working queue. Finally, we have a Map called sources, which will store data sources.

Now let's take a look at the onCreate() and onStart() methods for this class:

Listing 9-18. *onCreate() and onStart()*

```
 @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        LocalDataSource localData = new LocalDataSource(this.getResources());
        ARData.addMarkers(localData.getMarkers());

        NetworkDataSource twitter = new TwitterDataSource(this.getResources());
        sources.put("twitter", twitter);
        NetworkDataSource wikipedia = new
WikipediaDataSource(this.getResources());
        sources.put("wiki", wikipedia);
    }

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

```
        Location last = ARData.getCurrentLocation();
        updateData(last.getLatitude(), last.getLongitude(), last.getAltitude());
    }
```

In `onCreate()`, we begin by creating an instance of the `LocalDataSource` class and adding its markers to `ARData`. We then create a `NetworkDataSource` for both Twitter and Wikipedia, and add them to the sources Map. In `onStart()`, we get the last location data and update our data with it.

With this, we can now move onto the menu part of the code:

Listing 9-19. *Working with the menu*

```
@Override
    public boolean onCreateOptionsMenu(Menu menu) {
        MenuInflater inflater = getMenuInflater();
        inflater.inflate(R.menu.menu, menu);
        return true;
    }

    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        Log.v(TAG, "onOptionsItemSelected() item="+item);
        switch (item.getItemId()) {
            case R.id.showRadar:
                showRadar = !showRadar;
                item.setTitle(((showRadar)? "Hide" : "Show")+" Radar");
                break;
            case R.id.showZoomBar:
                showZoomBar = !showZoomBar;
                item.setTitle(((showZoomBar)? "Hide" : "Show")+" Zoom Bar");

zoomLayout.setVisibility((showZoomBar)?LinearLayout.VISIBLE:LinearLayout.GONE);
                break;
            case R.id.exit:
                finish();
                break;
        }
        return true;
    }
```

This is standard Android code, which we have used numerous times before. We simply create the menu from our XML menu resource, and then listen for clicks on the menus. For the Show Radar/Zoom bar options, we simply toggle their display, and for the exit option we, well, exit.

Now let's take a look at location updates and handling touches.

Listing 9-20. *Location Change and Touch Input*

```
   @Override
    public void onLocationChanged(Location location) {
        super.onLocationChanged(location);

        updateData(location.getLatitude(), location.getLongitude(),
location.getAltitude());
    }

 @Override
        protected void markerTouched(Marker marker) {
        Toast t = Toast.makeText(getApplicationContext(), marker.getName(),
Toast.LENGTH_SHORT);
        t.setGravity(Gravity.CENTER, 0, 0);
        t.show();
 }
```

When we get a new `location` object, we use it to update the data. We override the `markerTouched()` method to display a toast with the details of the marker that was touched.

Now let's take a look at this class's implementation of `updateDataOnZoom()`:

Listing 9-21. *updateDataOnZoom()*

```
@Override
        protected void updateDataOnZoom() {
            super.updateDataOnZoom();
        Location last = ARData.getCurrentLocation();
        updateData(last.getLatitude(),last.getLongitude(),last.getAltitude());
 }
```

In this implementation of `updateDataOnZoom()`, we get the location and then call `updateData()` and pass it the new location information.

Now let's take a look at the `updateData()` method:

Listing 9-22. *updateData()*

```
private void updateData(final double lat, final double lon, final double alt) {
        try {
            exeService.execute(
                new Runnable() {

                    public void run() {
                        for (NetworkDataSource source : sources.values())
                            download(source, lat, lon, alt);
                    }
                }
```

```
            );
        } catch (RejectedExecutionException rej) {
            Log.w(TAG, "Not running new download Runnable, queue is full.");
        } catch (Exception e) {
            Log.e(TAG, "Exception running download Runnable.",e);
        }
    }
```

In this method, we attempt to use a `Runnable` to download the data we need to show the Twitter posts and Wikipedia articles. If a `RejectedExecutionException` is encountered, a message to the LogCat is sent saying that the queue is full and cannot download the data right now. If any other exception is encountered, another message saying so is displayed in the Logcat.

Finally, we have the `download()` method:

Listing 9-23. *download()*

```
private static boolean download(NetworkDataSource source, double lat, double
lon, double alt) {
                if (source==null) return false;

                String url = null;
  try               {
                        url = source.createRequestURL(lat, lon, alt,
ARData.getRadius(), locale);
                } catch (NullPointerException e) {
   return                       false;
                }

                List<Marker> markers = null;
  try               {
   markers                       = source.parse(url);
                } catch (NullPointerException e) {
   return                       false;
                }

        ARData.addMarkers(markers);
        return true;
    }
}
```

In the `download` method, we first check to see whether the source is `null`. If it is, we return `false`. If it isn't `null`, we construct a URL to get the data. After this, we parse the result from the URL and store the data we get in the List markers. This data is then added to `ARData` via `ARData.addMarkers`.

This brings us to the end of all the Activities. We will now write the code for obtaining the data for the Twitter posts and the Wikipedia articles.

Getting the Data

To get the data, we will be creating five classes: the basic `DataSource` class, which will be extended by `LocalDataSource` and `NetworkDataSource`; `TwitterDataSource` and `WikipediaDataSource` will further extend `NetworkDataSource`.

Let's start with `DataSource`.

DataSource

`DataSource` is an exceptionally small abstract class:

Listing 9-24. *DataSource*

```
public abstract class DataSource {
    public abstract List<Marker> getMarkers();
}
```

There is simply only one member of this class: the `List getMarkers()`. This class is the base for all our other data classes.

Now let's take a look at the `LocalDataSource`.

LocalDataSource

`LocalDataSource` is used by `MainActivity` to add markers to `ARData`. The class is quite small.

Listing 9-25. *LocalDataSource*

```
public class LocalDataSource extends DataSource{
    private List<Marker> cachedMarkers = new ArrayList<Marker>();
    private static Bitmap icon = null;

    public LocalDataSource(Resources res) {
        if (res==null) throw new NullPointerException();

        createIcon(res);
    }

    protected void createIcon(Resources res) {
        if (res==null) throw new NullPointerException();

        icon=BitmapFactory.decodeResource(res, R.drawable.ic_launcher);
    }
```

```
    public List<Marker> getMarkers() {
        Marker atl = new IconMarker("ATL", 39.931269, -75.051261, 0,
Color.DKGRAY, icon);
        cachedMarkers.add(atl);

        Marker home = new Marker("Mt Laurel", 39.95, -74.9, 0, Color.YELLOW);
        cachedMarkers.add(home);

        return cachedMarkers;
    }
}
```

The constructor for this class takes a `Resource` object as an argument. It then calls `createIcon()`, which then assigns our app's default icon to the `icon` Bitmap. `getMarkers()`, well, gets the markers.

With this class done, let's take a look at `NetworkDataSource`.

NetworkDataSource

`NetworkDataSource` contains the basic setup for getting the data from our Twitter and Wikipedia sources.

Let's start with the class and global variable declarations:

Listing 9-26. *NetworkDataSource*

```
public abstract class NetworkDataSource extends DataSource {
    protected static final int MAX = 1000;
    protected static final int READ_TIMEOUT = 10000;
    protected static final int CONNECT_TIMEOUT = 10000;

    protected List<Marker> markersCache = null;

    public abstract String createRequestURL(double lat, double lon, double alt,
                                            float radius, String locale);

    public abstract List<Marker> parse(JSONObject root);
```

`MAX` specifies the maximum number of results to be displayed to the user. `READ_TIMEOUT` and `CONNECT_TIMEOUT` are the timeout values for the connection in milliseconds. `markersCache` is a List<markers> object that we will be using later on in this class. `createRequestURL` and `parse` are stubs that we will override in the extensions of this class.

Now let's take a look at the `getMarkers()` and `getHttpGETInputStream()` methods:

Listing 9-27. *getMarkers() and getHttpGETInputStream()*

```
public List<Marker> getMarkers() {
        return markersCache;
    }

    protected static InputStream getHttpGETInputStream(String urlStr) {
        if (urlStr == null)
            throw new NullPointerException();

        InputStream is = null;
        URLConnection conn = null;

        try {
            if (urlStr.startsWith("file://"))
                return new FileInputStream(urlStr.replace("file://", ""));

            URL url = new URL(urlStr);
            conn = url.openConnection();
            conn.setReadTimeout(READ_TIMEOUT);
            conn.setConnectTimeout(CONNECT_TIMEOUT);

            is = conn.getInputStream();

            return is;
        } catch (Exception ex) {
            try {
                is.close();
            } catch (Exception e) {
                // Ignore
            }
            try {
                if (conn instanceof HttpURLConnection)
                    ((HttpURLConnection) conn).disconnect();
            } catch (Exception e) {
                // Ignore
            }
            ex.printStackTrace();
        }
        return null;
    }
```

The `getMarkers()` method simply returns the `markersCache`. `getHttpGETInputStream()` is used to get an `InputStream` for the specified URL, which is passed to it as a `String`.

Now let's take a look at the `getHttpInputString()` and `parse()` methods:

Listing 9-28. *getHttpInputString() and parse()*

```
protected String getHttpInputString(InputStream is) {
        if (is == null)
            throw new NullPointerException();

        BufferedReader reader = new BufferedReader(new InputStreamReader(is),
                8 * 1024);
        StringBuilder sb = new StringBuilder();

        try {
            String line;
            while ((line = reader.readLine()) != null) {
                sb.append(line + "\n");
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            try {
                is.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        return sb.toString();
    }

    public List<Marker> parse(String url) {
        if (url == null)
            throw new NullPointerException();

        InputStream stream = null;
        stream = getHttpGETInputStream(url);
        if (stream == null)
            throw new NullPointerException();

        String string = null;
        string = getHttpInputString(stream);
        if (string == null)
            throw new NullPointerException();

        JSONObject json = null;
        try {
            json = new JSONObject(string);
        } catch (JSONException e) {
            e.printStackTrace();
        }
        if (json == null)
            throw new NullPointerException();
```

```
        return parse(json);
    }
}
```

In `getHttpInputString()`, we get the contents of the `InputStream` and put them in a `String`. In `parse()`, we get the JSON object from our data source and then call the other `parse()` method on it. In this class, the second `parse()` method is a stub, but it is overridden and implemented in the other classes.

Let's write up the two classes that extend `NetworkDataSource`: `TwitterDataSource` and `WikipediaDataSource`.

TwitterDataSource

`TwitterDataSource` extends `NetworkDataSource`. It is responsible for pulling the data about nearby tweets from Twitter's servers. `TwitterDataSource` has only two global variables to it.

Listing 9-29. *Declaring TwitterDataSource and its Global Variables*

```
public class TwitterDataSource extends NetworkDataSource {
        private static final String URL =
"http://search.twitter.com/search.json";
        private static Bitmap icon = null;
```

The string `URL` stores the base of the Twitter search URL. We will construct the parameters dynamically in `createRequestURL()`. The `icon` Bitmap, currently `null`, will store the Twitter logo in it. We will display this logo as the icon for each of our markers when showing Tweets.

Now let's take a look at the constructor, `createIcon()`, and `createRequestURL()` methods:

Listing 9-30. *The constructor, createIcon(), and createRequestURL()*

```
public TwitterDataSource(Resources res) {
        if (res==null) throw new NullPointerException();

 createIcon(res);
}

protected void createIcon(Resources res) {
        if (res==null) throw new NullPointerException();

 icon=BitmapFactory.decodeResource(res,        R.drawable.twitter);
}

@Override
```

```
public String createRequestURL(double lat, double lon, double alt, float radius,
String locale) {
        return URL+"?geocode=" + lat + "%2C" + lon + "%2C" + Math.max(radius,
1.0) + "km";
}
```

In the constructor, we take a `Resource` object as a parameter and then pass it to `createIcon()`. This is the exact same behavior as in `NetworkDataSource`. In `createIcon()`, we once again do the same thing we did in `NetworkDataSource`, except we use a different icon. Here we assign the Twitter drawable to the `icon` Bitmap, instead of the `ic_launcher` one. In `createRequestURL()`, we formulate a complete request URL for the Twitter JSON search application programming interface (API). The Twitter search API allows us to search tweets easily and anonymously. We supply the user's location in the geocode parameter and choose the bigger radius limit from the one that has been set by the user, and one kilometer.

Now we have two `parse()` methods and one `processJSONObject()`.

Listing 9-31. *The two parse() Methods and the processJSONObject() Method*

```
@Override
        public List<Marker> parse(String url) {
                if (url==null) throw new NullPointerException();

                InputStream stream = null;
        stream = getHttpGETInputStream(url);
        if (stream==null) throw new NullPointerException();

        String string = null;
        string = getHttpInputString(stream);
        if (string==null) throw new NullPointerException();

        JSONObject json = null;
        try {
                json = new JSONObject(string);
        } catch (JSONException e) {
            e.printStackTrace();
        }
        if (json==null) throw new NullPointerException();

        return parse(json);
}

@Override
        public List<Marker> parse(JSONObject root) {
                if (root==null) throw new NullPointerException();

                JSONObject jo = null;
```

```
                JSONArray dataArray = null;
        List<Marker> markers=new ArrayList<Marker>();

  try              {
   if(root.has("results"))                     dataArray =
root.getJSONArray("results");
                        if (dataArray == null) return markers;
                        int top = Math.min(MAX, dataArray.length());
                        for (int i = 0; i < top; i++) {

    jo                              = dataArray.getJSONObject(i);
    Marker                              ma = processJSONObject(jo);
    if(ma!=null)                              markers.add(ma);
                        }
                } catch (JSONException e) {
                    e.printStackTrace();
                }
  return                markers;
 }

        private Marker processJSONObject(JSONObject jo) {
                if (jo==null) throw new NullPointerException();

                if (!jo.has("geo")) throw new NullPointerException();

                Marker ma = null;
  try               {
   Double                        lat=null, lon=null;

   if(!jo.isNull("geo"))                        {
    JSONObject                               geo = jo.getJSONObject("geo");
    JSONArray                               coordinates =
geo.getJSONArray("coordinates");

 lat=Double.parseDouble(coordinates.getString(0));

 lon=Double.parseDouble(coordinates.getString(1));
   }                      else if(jo.has("location")) {
    Pattern                                 pattern = Pattern.compile("\\D*([0-
9.]+),\\s?([0-9.]+)");
    Matcher                                 matcher =
pattern.matcher(jo.getString("location"));

    if(matcher.find()){

 lat=Double.parseDouble(matcher.group(1));

 lon=Double.parseDouble(matcher.group(2));
                                }
                        }
```

```
   if(lat!=null)                        {
    String                               user=jo.getString("from_user");

    ma                                 = new IconMarker(
      user+":                                                  "+jo.getString("text"),
      lat,
      lon,
                                                        0,
      Color.RED,
      icon);
                        }
                } catch (Exception e) {
   e.printStackTrace();
                }
  return                ma;
 }
}
```

The first `parse()` method takes a URL in the form of a string parameter. The URL is then put through the `getHttpGETInputStream()` method, and the resulting Input Stream is passed to the `getHttpInputString()` method. Finally, a new `JSONObject` is created from the resulting String, and it is passed to the second `parse()` method.

In the second `parse()` method, we receive the `JSONObject` from the previous method as a parameter. We then first make sure that the object we received is not `null`. We then transfer the data from the object into a `JSONArray`, if it has any results in it. After this we need to loop through the array to create a marker for each of the results. To make sure we don't go outside of the array index, we find the smaller value between our maximum markers and the number of results. We then loop through the array, calling `processJSONObject()` to create each marker.

Finally, let's go through `processJSONObject()`. We once again start the method by checking if the `JSONObject` passed is `null`. After that, we check if the data in the object contains the `geo` data. If it doesn't, we don't use it because the `geo` data is important for placing it on the screen and radar. We then go through the `JSONObject` to get the coordinates of the tweet, the user, and the contents. All this data is then compiled into a marker, which is then returned to the second `parse()` method. The second `parse` method adds it to its `markers` List. Once all such markers have been created, the entire list is returned to the first `parse()` method, which further returns it to its caller in `MainActivity`.

Now let's take a look at the final Data Source class, `WikipediaDataSource`.

WikipediaDataSource

`WikipediaDataSource` is very similar to `TwitterDataSource` in structure and logic. The only major difference is in the parsing of the `JSONObject`.

Listing 9-32. *WikipediaDataSource*

```
public class WikipediaDataSource extends NetworkDataSource {
        private static final String BASE_URL =
"http://ws.geonames.org/findNearbyWikipediaJSON";

        private static Bitmap icon = null;

        public WikipediaDataSource(Resources res) {
            if (res==null) throw new NullPointerException();

        createIcon(res);
    }

    protected void createIcon(Resources res) {
        if (res==null) throw new NullPointerException();

        icon=BitmapFactory.decodeResource(res, R.drawable.wikipedia);
    }

 @Override
        public String createRequestURL(double lat, double lon, double alt, float
radius, String locale) {
  return               BASE_URL+
        "?lat=" + lat +
        "&lng=" + lon +
        "&radius="+ radius +
        "&maxRows=40" +
        "&lang=" + locale;

 }

 @Override
        public List<Marker> parse(JSONObject root) {
                if (root==null) return null;

                JSONObject jo = null;
                JSONArray dataArray = null;
        List<Marker> markers=new ArrayList<Marker>();

  try               {
   if(root.has("geonames"))                     dataArray =
root.getJSONArray("geonames");
                        if (dataArray == null) return markers;
```

```
    int                              top = Math.min(MAX, dataArray.length());
                                   for (int i = 0; i < top; i++) {

     jo                                      = dataArray.getJSONObject(i);
     Marker                                      ma = processJSONObject(jo);
     if(ma!=null)                                      markers.add(ma);
                                   }
                } catch (JSONException e) {
   e.printStackTrace();
                }
  return               markers;
 }

        private Marker processJSONObject(JSONObject jo) {
                if (jo==null) return null;

        Marker ma = null;
        if (    jo.has("title") &&
                        jo.has("lat") &&
                        jo.has("lng") &&
                        jo.has("elevation")
        ) {
                try {
                        ma = new IconMarker(
                                        jo.getString("title"),
                                        jo.getDouble("lat"),
                                        jo.getDouble("lng"),
                                        jo.getDouble("elevation"),
                                        Color.WHITE,
                                        icon);
                } catch (JSONException e) {
                        e.printStackTrace();
                }
        }
        return ma;
 }
}
```

Unlike Twitter, Wikipedia does not provide an official search facility. However, geonames.org provides a JSON-based search for Wikipedia, and we will be using it. The next difference from the `TwitterDataSource` is the icon. We just use a different `drawable` when creating the icon. The basic code for getting and parsing the `JSONObject` is the same; only the values are different.

We will now write to classes that help us with positioning the markers on the screen and in real life.

Positioning Classes

We need a set of classes that handle the position related work for our app. These classes handle the user's physical location, and the positiong of objects on the screen.

PhysicalLocationUtility

This class is used to represent the user's location in the real world in three dimensions.

Listing 9-33. *PhysicalLocationUtility.java*

```
public class PhysicalLocationUtility {
        private double latitude = 0.0;
        private double longitude = 0.0;
        private double altitude = 0.0;

        private static float[] x = new float[1];
        private static double y = 0.0d;
        private static float[] z = new float[1];

        public PhysicalLocationUtility() { }

        public PhysicalLocationUtility(PhysicalLocationUtility pl) {
                if (pl==null) throw new NullPointerException();

 set(pl.latitude,               pl.longitude, pl.altitude);
}

        public void set(double latitude, double longitude, double altitude) {
        this.latitude = latitude;
        this.longitude = longitude;
        this.altitude = altitude;
}

        public void setLatitude(double latitude) {
 this.latitude               = latitude;
}

        public double getLatitude() {
 return               latitude;
}

        public void setLongitude(double longitude) {
 this.longitude               = longitude;
}
```

```
        public double getLongitude() {
 return                longitude;
 }

        public void setAltitude(double altitude) {
 this.altitude                = altitude;
 }

        public double getAltitude() {
 return                altitude;
 }

        public static synchronized void convLocationToVector(Location org,
PhysicalLocationUtility gp, Vector v) {
                if (org==null || gp==null || v==null)
                    throw new NullPointerException("Location,
PhysicalLocationUtility, and Vector cannot be NULL.");

  Location.distanceBetween(                    org.getLatitude(),
org.getLongitude(),

        gp.getLatitude(), org.getLongitude(),
         z);

  Location.distanceBetween(                    org.getLatitude(),
org.getLongitude(),

        org.getLatitude(), gp.getLongitude(),
         x);
                y = gp.getAltitude() - org.getAltitude();
                if (org.getLatitude() < gp.getLatitude())
   z[0]                      *= -1;
                if (org.getLongitude() > gp.getLongitude())
   x[0]                      *= -1;

                v.set(x[0], (float) y, z[0]);
 }

 @Override
        public String toString() {
                return "(lat=" + latitude + ", lng=" + longitude + ", alt=" +
altitude + ")";
 }
}
```

The first three doubles are used to store the `latitude`, `longitude` and `altitude` respectively, as their names suggest. The x, y, and z store the final three-dimensional position data. The `setLatitude()`, `setLongitude()`, and `setAltitude()` methods set the latitude, longitude, and altitude, respectively.

Their `get()` counterparts simply return the current value. The `convLocationToVector()` methods convert the location to a `Vector`. We will write a `Vector` class later on in this chapter. The `toString()` method simply compiles the latitude, longitude, and altitude into a String and returns it to the calling method.

Now let's take a look at `ScreenPositionUtility`.

ScreenPositionUtility

`ScreenPositionUtility` is used when displaying the lines for the radar.

Listing 9-34. *ScreenPositionUtility.java*

```
public class ScreenPositionUtility {
    private float x = 0f;
    private float y = 0f;

        public ScreenPositionUtility() {
        set(0, 0);
    }

    public void set(float x, float y) {
        this.x = x;
        this.y = y;
    }

    public float getX() {
 return               x;
}

        public void setX(float x) {
 this.x               = x;
}

        public float getY() {
 return               y;
}

        public void setY(float y) {
 this.y               = y;
}

    public void rotate(double t) {
        float xp = (float) Math.cos(t) * x - (float) Math.sin(t) * y;
        float yp = (float) Math.sin(t) * x + (float) Math.cos(t) * y;

        x = xp;
```

```
        y = yp;
    }

    public void add(float x, float y) {
        this.x += x;
        this.y += y;
    }

    @Override
    public String toString() {
        return "x="+x+" y="+y;
    }
}
```

The `setX()` and `setY()` methods set the values for the x and y variable floats, respectively. The `set()` method sets the values for both x and y together. The `getX()` and `getY()` methods simply return the values of x and y. The `rotate()` method rotates the x and y values around the angle `t`. The `add()` method adds the passed values to x and y, respectively. Finally, the `toString()` method returns a string with the value of both x and y in it.

Now let's take a look at the UI code.

The UI Works

PaintableObject

`PaintableObject` is the base class for all our custom bits of the user interface. Some of its methods are just stubs that we override in its subclasses. This class contains a lot of methods for drawing certain objects such as lines, bitmaps, points, etc. on a given canvas.

Listing 9-35. *PaintableObject.java*

```
public abstract class PaintableObject {
    private Paint paint = new Paint(Paint.ANTI_ALIAS_FLAG);

    public PaintableObject() {
        if (paint==null) {
            paint = new Paint();
            paint.setTextSize(16);
            paint.setAntiAlias(true);
            paint.setColor(Color.BLUE);
            paint.setStyle(Paint.Style.STROKE);
        }
    }
```

```
    public abstract float getWidth();

    public abstract float getHeight();

    public abstract void paint(Canvas canvas);

    public void setFill(boolean fill) {
        if (fill)
            paint.setStyle(Paint.Style.FILL);
        else
            paint.setStyle(Paint.Style.STROKE);
    }

    public void setColor(int c) {
        paint.setColor(c);
    }

    public void setStrokeWidth(float w) {
        paint.setStrokeWidth(w);
    }

    public float getTextWidth(String txt) {
        if (txt==null) throw new NullPointerException();
        return paint.measureText(txt);
    }

    public float getTextAsc() {
        return -paint.ascent();
    }

    public float getTextDesc() {
        return paint.descent();
    }

    public void setFontSize(float size) {
        paint.setTextSize(size);
    }

    public void paintLine(Canvas canvas, float x1, float y1, float x2, float y2)
{
        if (canvas==null) throw new NullPointerException();

        canvas.drawLine(x1, y1, x2, y2, paint);
    }

    public void paintRect(Canvas canvas, float x, float y, float width, float
height) {
        if (canvas==null) throw new NullPointerException();
```

```
        canvas.drawRect(x, y, x + width, y + height, paint);
    }

    public void paintRoundedRect(Canvas canvas, float x, float y, float width,
float height) {
        if (canvas==null) throw new NullPointerException();

        RectF rect = new RectF(x, y, x + width, y + height);
        canvas.drawRoundRect(rect, 15F, 15F, paint);
    }

    public void paintBitmap(Canvas canvas, Bitmap bitmap, Rect src, Rect dst) {
        if (canvas==null || bitmap==null) throw new NullPointerException();

        canvas.drawBitmap(bitmap, src, dst, paint);
    }

    public void paintBitmap(Canvas canvas, Bitmap bitmap, float left, float top)
{
        if (canvas==null || bitmap==null) throw new NullPointerException();

        canvas.drawBitmap(bitmap, left, top, paint);
    }

    public void paintCircle(Canvas canvas, float x, float y, float radius) {
        if (canvas==null) throw new NullPointerException();

        canvas.drawCircle(x, y, radius, paint);
    }

    public void paintText(Canvas canvas, float x, float y, String text) {
        if (canvas==null || text==null) throw new NullPointerException();

        canvas.drawText(text, x, y, paint);
    }

    public void paintObj(       Canvas canvas, PaintableObject obj,
                                               float x, float y,
                                               float rotation, float scale)
    {
        if (canvas==null || obj==null) throw new NullPointerException();

        canvas.save();
        canvas.translate(x+obj.getWidth()/2, y+obj.getHeight()/2);
        canvas.rotate(rotation);
        canvas.scale(scale,scale);
        canvas.translate(-(obj.getWidth()/2), -(obj.getHeight()/2));
        obj.paint(canvas);
        canvas.restore();
    }
```

```
    public void paintPath(      Canvas canvas, Path path,
                                               float x, float y, float width,
                                               float height, float rotation,
float scale)
    {
        if (canvas==null || path==null) throw new NullPointerException();

        canvas.save();
        canvas.translate(x + width / 2, y + height / 2);
        canvas.rotate(rotation);
        canvas.scale(scale, scale);
        canvas.translate(-(width / 2), -(height / 2));
        canvas.drawPath(path, paint);
        canvas.restore();
    }
}
```

The entire class has only one global variable: a `paint` object with anti-aliasing enabled. Anti-aliasing smoothes out the lines of the object being drawn. The constructor initializes the `paint` object by setting the text size to 16, enabling anti-aliasing, setting the paint color to blue, and setting the paint style to `Paint.Style.STROKE`.

The following three methods—`getWidth()`, `getHeight()`, and `paint()`—are left as method stubs to be overridden if required.

The `setFill()` method allows us to change the paint style to `Paint.Style.FILL` or `Paint.Style.STROKE`. The `setColor()` method sets the color of the paint to the color corresponding to the integer it takes as an argument. The `setStrokeWidth()` allows us to set the width of the paint's stroke. `getTextWidth()` returns the width of the text it takes as an argument. `getTextAsc()` and `getTextDesc()` return the ascent and descent of the text, respectively. `setFontSize()` allows us to set the font size of the paint. All the remaining methods with paint prefixed to their names draw the object that is written after the paint on the supplied `canvas`. For example, `paintLine()` draws a line with the supplied coordinates on the supplied `canvas`.

Now let's take a look at the set of classes that extends `PaintableObject`.

PaintableBox

The `PaintableBox` class allows us to draw a box outline. It's a simple class and not very big.

Listing 9-36. *PaintableBox.java*

```
public class PaintableBox extends PaintableObject {
    private float width=0, height=0;
        private int borderColor = Color.rgb(255, 255, 255);
        private int backgroundColor = Color.argb(128, 0, 0, 0);

        public PaintableBox(float width, float height) {
                this(width, height, Color.rgb(255, 255, 255), Color.argb(128, 0,
0, 0));
 }

        public PaintableBox(float width, float height, int borderColor, int
bgColor) {
                set(width, height, borderColor, bgColor);
 }

    public void set(float width, float height) {
        set(width, height, borderColor, backgroundColor);
    }

        public void set(float width, float height, int borderColor, int bgColor)
{
            this.width = width;
            this.height = height;
            this.borderColor = borderColor;
  this.backgroundColor               = bgColor;
 }

 @Override
        public void paint(Canvas canvas) {
                if (canvas==null) throw new NullPointerException();

  setFill(true);
  setColor(backgroundColor);
                paintRect(canvas, 0, 0, width, height);

  setFill(false);
  setColor(borderColor);
                paintRect(canvas, 0, 0, width, height);
 }

 @Override
        public float getWidth() {
  return               width;
 }

 @Override
        public float getHeight() {
  return               height;
```

```
 }
}
```

This class has two constructors, one of which calls the other. The reason is that one of the constructors allows you to set only the width and height of the box, while the second one allows you to set its colors as well. When calling the first one, it uses the supplied width and height to call the second one with the default colors. The second constructor then calls the second `set()` method to set those values. The `paint()` method simply draws the box on the specified `canvas`. `getWidth()` and `getHeight()` just return the width and height of the box.

Now let's take a look at the `PaintableBoxedText`.

PaintableBoxedText

`PaintableBoxedText` draws text on the canvas with a box around it.

Listing 9-37. *PaintableBox.java*

```
public class PaintableBoxedText extends PaintableObject {
    private float width=0, height=0;
        private float areaWidth=0, areaHeight=0;
        private ArrayList<String> lineList = null;
        private String[] lines = null;
        private float[] lineWidths = null;
        private float lineHeight = 0;
        private float maxLineWidth = 0;
        private float pad = 0;

        private String txt = null;
    private float fontSize = 12;
        private int borderColor = Color.rgb(255, 255, 255);
        private int backgroundColor = Color.argb(160, 0, 0, 0);
        private int textColor = Color.rgb(255, 255, 255);

        public PaintableBoxedText(String txtInit, float fontSizeInit, float
maxWidth) {
                this(txtInit, fontSizeInit, maxWidth, Color.rgb(255, 255, 255),
Color.argb(128, 0, 0, 0), Color.rgb(255, 255, 255));
 }

        public PaintableBoxedText(String txtInit, float fontSizeInit, float
maxWidth, int borderColor, int bgColor, int textColor) {
                set(txtInit, fontSizeInit, maxWidth, borderColor, bgColor,
textColor);
 }
```

```
        public void set(String txtInit, float fontSizeInit, float maxWidth, int
borderColor, int bgColor, int textColor) {
                if (txtInit==null) throw new NullPointerException();

  this.borderColor               = borderColor;
  this.backgroundColor               = bgColor;
  this.textColor               = textColor;
  this.pad               = getTextAsc();

  set(txtInit,               fontSizeInit, maxWidth);
 }

        public void set(String txtInit, float fontSizeInit, float maxWidth) {
                if (txtInit==null) throw new NullPointerException();

  try               {
   prepTxt(txtInit,                      fontSizeInit, maxWidth);
                } catch (Exception ex) {
   ex.printStackTrace();
                        prepTxt("TEXT PARSE ERROR", 12, 200);
                }
 }

        private void prepTxt(String txtInit, float fontSizeInit, float maxWidth)
{
                if (txtInit==null) throw new NullPointerException();

  setFontSize(fontSizeInit);

  txt               = txtInit;
  fontSize               = fontSizeInit;
                areaWidth = maxWidth - pad;
                lineHeight = getTextAsc() + getTextDesc();

                if (lineList==null) lineList = new ArrayList<String>();
  else               lineList.clear();

                BreakIterator boundary = BreakIterator.getWordInstance();
  boundary.setText(txt);

                int start = boundary.first();
                int end = boundary.next();
                int prevEnd = start;
                while (end != BreakIterator.DONE) {
                        String line = txt.substring(start, end);
                        String prevLine = txt.substring(start, prevEnd);
   float                      lineWidth = getTextWidth(line);

                        if (lineWidth > areaWidth) {
    if(prevLine.length()>0)                                lineList.add(prevLine);
```

```
    start                                = prevEnd;
                        }

   prevEnd                        = end;
   end                         = boundary.next();
                }
                String line = txt.substring(start, prevEnd);
  lineList.add(line);

                if (lines==null || lines.length!=lineList.size()) lines = new
String[lineList.size()];
                if (lineWidths==null || lineWidths.length!=lineList.size())
lineWidths = new float[lineList.size()];
  lineList.toArray(lines);

  maxLineWidth                = 0;
                for (int i = 0; i < lines.length; i++) {
   lineWidths[i]                          = getTextWidth(lines[i]);
   if                         (maxLineWidth < lineWidths[i])
    maxLineWidth                                = lineWidths[i];
                }
  areaWidth                = maxLineWidth;
                areaHeight = lineHeight * lines.length;

                width = areaWidth + pad * 2;
                height = areaHeight + pad * 2;
 }

 @Override
        public void paint(Canvas canvas) {
                if (canvas==null) throw new NullPointerException();

            setFontSize(fontSize);

  setFill(true);
  setColor(backgroundColor);
                paintRoundedRect(canvas, 0, 0, width, height);

  setFill(false);
  setColor(borderColor);
                paintRoundedRect(canvas, 0, 0, width, height);

                for (int i = 0; i < lines.length; i++) {
   String                         line = lines[i];
   setFill(true);
   setStrokeWidth(0);
   setColor(textColor);
                        paintText(canvas, pad, pad + lineHeight * i +
getTextAsc(), line);
```

```
                }
 }

 @Override
        public float getWidth() {
  return               width;
 }

 @Override
        public float getHeight() {
  return               height;
 }
}
```

Once again, there are two constructors that do the same thing as those in `PaintableBox`. The first major difference from `PaintableBox` is that of the new method `prepTxt()`. `prepTxt()` prepares text by cutting it into different lines sized to fit into the box, instead of having one long string that happily leaves the box and overflows outside. The `paint()` method then draws the basic box first and then uses a `for` loop to add each of the lines to it.

Now let's take a look at `PaintableCircle`.

PaintableCircle

`PaintableCircle` allows us to, well, paint a circle onto the supplied `Canvas`. It's a simple class, with simple code:

Listing 9-38. *PaintableCircle.java*

```
public class PaintableCircle extends PaintableObject {
    private int color = 0;
    private float radius = 0;
    private boolean fill = false;

    public PaintableCircle(int color, float radius, boolean fill) {
        set(color, radius, fill);
    }

    public void set(int color, float radius, boolean fill) {
        this.color = color;
        this.radius = radius;
        this.fill = fill;
    }

    @Override
    public void paint(Canvas canvas) {
        if (canvas==null) throw new NullPointerException();
```

```
        setFill(fill);
        setColor(color);
        paintCircle(canvas, 0, 0, radius);
    }

    @Override
    public float getWidth() {
        return radius*2;
    }

    @Override
    public float getHeight() {
        return radius*2;
    }
}
```

This time, there is only one constructor that allows us to set the radius and the colors of the circle. There is also only one `set()` method, which is called by the constructor. The `paint()` method draws a circle with the specified properties on the given `canvas`. The `getWidth()` and `getHeight()` methods return the diameter because this is a circle.

Now let's take a look at `PaintableGps`.

PaintableGps

`PaintableGps` is a lot like `PaintableCircle`, except that it also allows us to set the stroke width of the circle being drawn.

Listing 9-39. *PaintableGps.java*

```
public class PaintableGps extends PaintableObject {
    private float radius = 0;
    private float strokeWidth = 0;
    private boolean fill = false;
    private int color = 0;

    public PaintableGps(float radius, float strokeWidth, boolean fill, int
color) {
        set(radius, strokeWidth, fill, color);
    }

    public void set(float radius, float strokeWidth, boolean fill, int color) {
        this.radius = radius;
        this.strokeWidth = strokeWidth;
        this.fill = fill;
        this.color = color;
```

```
    }

 @Override
    public void paint(Canvas canvas) {
        if (canvas==null) throw new NullPointerException();

        setStrokeWidth(strokeWidth);
        setFill(fill);
        setColor(color);
        paintCircle(canvas, 0, 0, radius);
    }

 @Override
    public float getWidth() {
        return radius*2;
    }

 @Override
    public float getHeight() {
            return radius*2;
    }
}
```

Once more, there is only one constructor, which calls the `set()` method to set the colors, size of the circle being drawn and the width of the stroke. The `paint()` method draws the circle with the specified properties on the supplied `Canvas` as usual. Once again, `getWidth()` and `getHeight()` return the circle's diameter.

Now let's take a look at `PaintableIcon`.

PaintableIcon

We use `PaintableIcon` to draw the icons for Twitter and Wikipedia.

Listing 9-40. *PaintableIcon.java*

```
    public Class PaintableIcon Extends PaintableObject {
    private Bitmap bitmap=null;

    public PaintableIcon(Bitmap bitmap, int width, int height) {
        set(bitmap,width,height);
    }

    public void set(Bitmap bitmap, int width, int height) {
        if (bitmap==null) throw new NullPointerException();

        this.bitmap = Bitmap.createScaledBitmap(bitmap, width, height, true);
```

```
    }

 @Override
    public void paint(Canvas canvas) {
        if (canvas==null || bitmap==null) throw new NullPointerException();

        paintBitmap(canvas, bitmap, -(bitmap.getWidth()/2), -
(bitmap.getHeight()/2));
    }

 @Override
    public float getWidth() {
        return bitmap.getWidth();
    }

 @Override
    public float getHeight() {
        return bitmap.getHeight();
    }
}
```

The constructor takes the bitmap to be drawn, along with the width and height according to which it is to be drawn and then calls the `set()` method to set them. In the `set()` method, the `Bitmap` is scaled to the size specified. The `paint()` method then draws it onto the supplied `canvas`. The `getWidth()` and `getHeight()` methods return the width and height of the bitmap we drew in the end, not the bitmap that was passed in the constructor.

Now we'll take a look at creating a class that paints lines.

PaintableLine

`PaintableLine` allows us to paint a line in a specified color onto a supplied `Canvas`.

Listing 9-41. *PaintableLine.java*

```
public class PaintableLine extends PaintableObject {
    private int color = 0;
    private float x = 0;
    private float y = 0;

    public PaintableLine(int color, float x, float y) {
        set(color, x, y);
    }

    public void set(int color, float x, float y) {
        this.color = color;
```

```
        this.x = x;
        this.y = y;
    }

 @Override
    public void paint(Canvas canvas) {
        if (canvas==null) throw new NullPointerException();

        setFill(false);
        setColor(color);
        paintLine(canvas, 0, 0, x, y);
    }

 @Override
    public float getWidth() {
        return x;
    }

 @Override
    public float getHeight() {
        return y;
    }
}
```

The constructor takes the color and X and Y points of the line and passes them to `set()` to set them. The `paint()` method draws it on the `Canvas` using `PaintableObject.` `getWidth()` and `getHeight()` return x and y, respectively.

Now we'll take a look at `PaintablePoint`.

PaintablePoint

`PaintablePoint` is used to draw a single point on a single canvas. It is used when making our radar.

Listing 9-42. *PaintablePoint*

```
public class PaintablePoint extends PaintableObject {
    private static int width=2;
    private static int height=2;
    private int color = 0;
    private boolean fill = false;

    public PaintablePoint(int color, boolean fill) {
        set(color, fill);
    }

    public void set(int color, boolean fill) {
```

```
        this.color = color;
        this.fill = fill;
    }

 @Override
    public void paint(Canvas canvas) {
        if (canvas==null) throw new NullPointerException();

        setFill(fill);
        setColor(color);
        paintRect(canvas, -1, -1, width, height);
    }

 @Override
    public float getWidth() {
        return width;
    }

 @Override
    public float getHeight() {
        return height;
    }
}
```

The point being drawn is not actually a point; it is just a really small rectangle. The constructor accepts the colors and sets them via the `set()` method, and then `paint()` draws a really small rectangle. `getWidth()` and `getHeight()` simply return the width and height of the rectangle being drawn.

Now let's take a look at `PaintablePosition`.

PaintablePosition

`PaintablePosition` extends `PaintableObject` and adds the ability to rotate and scale the thing being painted.

Listing 9-43. *PaintablePosition*

```
public class PaintablePosition extends PaintableObject {
    private float width=0, height=0;
    private float objX=0, objY=0, objRotation=0, objScale=0;
    private PaintableObject obj = null;

    public PaintablePosition(PaintableObject drawObj, float x, float y, float
rotation, float scale) {
        set(drawObj, x, y, rotation, scale);
    }
```

```
    public void set(PaintableObject drawObj, float x, float y, float rotation,
float scale) {
        if (drawObj==null) throw new NullPointerException();

        this.obj = drawObj;
        this.objX = x;
        this.objY = y;
        this.objRotation = rotation;
        this.objScale = scale;
        this.width = obj.getWidth();
        this.height = obj.getHeight();
    }

    public void move(float x, float y) {
        objX = x;
        objY = y;
    }

    public float getObjectsX() {
        return objX;
    }

    public float getObjectsY() {
        return objY;
    }

 @Override
    public void paint(Canvas canvas) {
        if (canvas==null || obj==null) throw new NullPointerException();

        paintObj(canvas, obj, objX, objY, objRotation, objScale);
    }

 @Override
    public float getWidth() {
        return width;
    }

 @Override
    public float getHeight() {
        return height;
    }

    @Override
        public String toString() {
            return "objX="+objX+" objY="+objY+" width="+width+" height="+height;
 }
}
```

The constructor takes an instance of the `PaintableObject` class, the X and Y coordinates of the position, the rotation angle, and the scale amount. The constructor then passes all this data to the `set()` method, which sets the values. We now have three new methods more than the others that have been there in the classes that have extended `PaintableObject` so far: `move()`, `getObjectsX()`, and `getObjectsY()`. `getObjectsX()` and `getObjectsY()` return the x and y values that were passed to the constructor, respectively. `move()` allows us to move the object to new X and Y coordinates. The `paint()` method once again draws the object on the supplied `canvas`. `getWidth()` and `getHeight()` return the width and the height of the object. `toString()` returns X coordinate, Y coordinate, width, and height of the object in a single string.

Now let's take a look at `PaintableRadarPoints`.

PaintableRadarPoints

`PaintableRadarPoints` is used to draw all the markers' relative positions on the radar.

Listing 9-44. *PaintableRadarPoints*

```
public class PaintableRadarPoints extends PaintableObject {
    private final float[] locationArray = new float[3];
        private PaintablePoint paintablePoint = null;
        private PaintablePosition pointContainer = null;

 @Override
    public void paint(Canvas canvas) {
                if (canvas==null) throw new NullPointerException();

                float range = ARData.getRadius() * 1000;
                float scale = range / Radar.RADIUS;
                for (Marker pm : ARData.getMarkers()) {
                    pm.getLocation().get(locationArray);
                    float x = locationArray[0] / scale;
                    float y = locationArray[2] / scale;
                    if ((x*x+y*y)<(Radar.RADIUS*Radar.RADIUS)) {
                        if (paintablePoint==null) paintablePoint = new
PaintablePoint(pm.getColor(),true);
                        else paintablePoint.set(pm.getColor(),true);

                        if (pointContainer==null) pointContainer = new
PaintablePosition(      paintablePoint,
                                (x+Radar.RADIUS-1),
                                (y+Radar.RADIUS-1),
                                0,
```

```
                        1);
                    else pointContainer.set(paintablePoint,
                        (x+Radar.RADIUS-1),
                        (y+Radar.RADIUS-1),
                        0,
                        1);

                    pointContainer.paint(canvas);
                }
            }
    }

 @Override
    public float getWidth() {
        return Radar.RADIUS * 2;
    }

 @Override
    public float getHeight() {
        return Radar.RADIUS * 2;
    }
}
```

In this class, there is no constructor. Instead, only the `paint()`, `getWidth()`, and `getHeight()` methods are present. `getWidth()` and `getHeight()` return the diameter of the point we draw to represent a marker. In the `paint()` method, we use a `for` loop to draw a dot on the radar for every marker.

Now let's take a look at `PaintableText`.

PaintableText

`PaintableText` is an extension of `PaintableObject` that draws text. We use it to display the text on the radar:

Listing 9-45. *PaintableText*

```
public class PaintableText extends PaintableObject {
    private static final float WIDTH_PAD = 4;
    private static final float HEIGHT_PAD = 2;

    private String text = null;
    private int color = 0;
    private int size = 0;
    private float width = 0;
    private float height = 0;
    private boolean bg = false;
```

```
    public PaintableText(String text, int color, int size, boolean
paintBackground) {
        set(text, color, size, paintBackground);
    }

    public void set(String text, int color, int size, boolean paintBackground) {
        if (text==null) throw new NullPointerException();

        this.text = text;
        this.bg = paintBackground;
        this.color = color;
        this.size = size;
        this.width = getTextWidth(text) + WIDTH_PAD * 2;
        this.height = getTextAsc() + getTextDesc() + HEIGHT_PAD * 2;
    }

 @Override
    public void paint(Canvas canvas) {
        if (canvas==null || text==null) throw new NullPointerException();

        setColor(color);
        setFontSize(size);
        if (bg) {
            setColor(Color.rgb(0, 0, 0));
            setFill(true);
            paintRect(canvas, -(width/2), -(height/2), width, height);
            setColor(Color.rgb(255, 255, 255));
            setFill(false);
            paintRect(canvas, -(width/2), -(height/2), width, height);
        }
        paintText(canvas, (WIDTH_PAD - width/2), (HEIGHT_PAD + getTextAsc() -
height/2), text);
    }

 @Override
    public float getWidth() {
        return width;
    }

 @Override
    public float getHeight() {
        return height;
    }
}
```

This class's constructor takes the text, its color, its size, and its background color as arguments and then passes all that to the `set()` method to be set. The `paint()` method draws the text, along with its background color. The `getWidth()` and `getHeight()` once again return the width and height.

Now before we get to the main UI components such as the `Radar` class, `Marker` class, and `IconMarker` class, we need to create some utility classes.

Utility Classes

In our app, we have some Utility classes. These classes handle the Vector and Matrix functions, implement the LowPassFilter and also handle the calculation of values like the pitch.

Vector

The first of our utility classes is the `Vector` class. This class handles the maths behind `Vectors`. We had a similar class called `Vector3D` in the previous chapter, but this one is far more comprehensive. All of it is pure Java, with nothing Android-specific in it. The code is adapted from the `Vector` class of the free and open source Mixare framework (http://www.mixare.org/). We'll look at the methods grouped by type, such as mathematical functions, setting values, and so on.

First, let's take a look at the global variables and the constructors.

Listing 9-46. *Vector's Constructors and Global Variables*

```
public class Vector {
    private final float[] matrixArray = new float[9];

        private volatile float x = 0f;
    private volatile float y = 0f;
        private volatile float z = 0f;

        public Vector() {
  this(0,               0, 0);
 }

        public Vector(float x, float y, float z) {
            set(x, y, z);
 }
```

`matrixArray` is an array that we use in the `prod()` method later on in this class. The floats `x`, `y`, and `z` are the three values of any given `Vector`. The first constructor creates a `Vector` with `x`, `y`, and `z`, all set to zero. The second constructor sets `x`, `y`, and `z` to the supplied values.

Now let's take a look at the getter and setter methods for this class:

Listing 9-47. *get() and set()*

```
        public synchronized float getX() {
        return x;
    }
    public synchronized void setX(float x) {
        this.x = x;
    }

    public synchronized float getY() {
        return y;
    }

    public synchronized void setY(float y) {
        this.y = y;
    }

    public synchronized float getZ() {
        return z;
    }

    public synchronized void setZ(float z) {
        this.z = z;
    }

    public synchronized void get(float[] array) {
        if (array==null || array.length!=3)
            throw new IllegalArgumentException("get() array must be non-NULL and
size of 3");

        array[0] = this.x;
        array[1] = this.y;
        array[2] = this.z;
    }

    public void set(Vector v) {
        if (v==null) return;

        set(v.x, v.y, v.z);
    }

    public void set(float[] array) {
        if (array==null || array.length!=3)
            throw new IllegalArgumentException("get() array must be non-NULL and
size of 3");

        set(array[0], array[1], array[2]);
    }

        public synchronized void set(float x, float y, float z) {
```

```
 this.x              = x;
 this.y              = y;
 this.z              = z;
}
```

getX(), getY(), and getZ() return the values of x, y, and z to the calling method, respectively, while their set() counterparts update said value. The get() method that takes a float array as an argument will give you the values of x, y, and z in one go. Out of the remaining three set() methods, two of them end up calling set(float x, float y, float z), which sets the values for x, y, and z. The other two set() methods that call this one simply exist to allow us to set the values using an array or pre-existing Vector, instead of always having to pass individual values for x, y, and z.

Now we'll move onto the maths part of this class:

Listing 9-48. *The maths Part of the Vector Class*

```
@Override
       public synchronized boolean equals(Object obj) {
               if (obj==null) return false;

               Vector v = (Vector) obj;
               return (v.x == this.x && v.y == this.y && v.z == this.z);
}

       public synchronized void add(float x, float y, float z) {
 this.x              += x;
 this.y              += y;
 this.z              += z;
}

       public void add(Vector v) {
 if              (v==null) return;

 add(v.x,              v.y, v.z);
}

       public void sub(Vector v) {
 if              (v==null) return;

 add(-v.x,              -v.y, -v.z);
}

       public synchronized void mult(float s) {
           this.x *= s;
           this.y *= s;
           this.z *= s;
}
```

```
        public synchronized void divide(float s) {
            this.x /= s;
            this.y /= s;
            this.z /= s;
 }

        public synchronized float length() {
                return (float) Math.sqrt(this.x * this.x + this.y * this.y +
this.z * this.z);
 }

        public void norm() {
  divide(length());
 }

        public synchronized void cross(Vector u, Vector v) {
                if (v==null || u==null) return;

                float x = u.y * v.z - u.z * v.y;
                float y = u.z * v.x - u.x * v.z;
                float z = u.x * v.y - u.y * v.x;
  this.x                = x;
  this.y                = y;
  this.z                = z;
 }

        public synchronized void prod(Matrix m) {
  if               (m==null) return;

  m.get(matrixArray);
        float xTemp = matrixArray[0] * this.x + matrixArray[1] * this.y +
matrixArray[2] * this.z;
        float yTemp = matrixArray[3] * this.x + matrixArray[4] * this.y +
matrixArray[5] * this.z;
        float zTemp = matrixArray[6] * this.x + matrixArray[7] * this.y +
matrixArray[8] * this.z;

  this.x                = xTemp;
  this.y                = yTemp;
  this.z                = zTemp;
 }

 @Override
        public synchronized String toString() {
                return "x = " + this.x + ", y =  " + this.y + ", z = " + this.z;
 }
}
```

The `equals()` method compares the vector to a given object to see whether they are equal. The `add()` and `sub()` methods add and subtract the arguments to and from the vector, respectively. The `mult()` method multiplies all the values by the passed float. The `divide()` method divides all the values by the passed float. The `length()` method returns the length of the Vector. The `norm()` method divides the Vector by its length. The `cross()` method cross multiplies two Vectors. The `prod()` method multiplies the Vector with the supplied Matrix. `toString()` returns the values of `x`, `y`, and `z` in a human readable format.

Next we have the `Utilities` class.

Utilities

The `Utilities` class contains a single `getAngle()` method that we use to get the angle when calculating stuff like the pitch in `PitchAzimuthCalculator`. The math in it is simple trigonometry.

Listing 9-49. *Utilities*

```
public abstract class Utilities {

    private Utilities() { }

    public static final float getAngle(float center_x, float center_y, float
post_x, float post_y) {
        float tmpv_x = post_x - center_x;
        float tmpv_y = post_y - center_y;
        float d = (float) Math.sqrt(tmpv_x * tmpv_x + tmpv_y * tmpv_y);
        float cos = tmpv_x / d;
        float angle = (float) Math.toDegrees(Math.acos(cos));

        angle = (tmpv_y < 0) ? angle * -1 : angle;

        return angle;
    }
}
```

Now let's take a look at that `PitchAzimuthCalculator`.

PitchAzimuthCalculator

`PitchAzimuthCalculator` is a class that is used to calculate the pitch and azimuth when given a matrix:

Listing 9-50. *PitchAzimuthCalculator*

```
public class PitchAzimuthCalculator {
    private static final Vector looking = new Vector();
    private static final float[] lookingArray = new float[3];

    private static volatile float azimuth = 0;

    private static volatile float pitch = 0;

    private PitchAzimuthCalculator() {};

    public static synchronized float getAzimuth() {
        return PitchAzimuthCalculator.azimuth;
    }
    public static synchronized float getPitch() {
        return PitchAzimuthCalculator.pitch;
    }

    public static synchronized void calcPitchBearing(Matrix rotationM) {
        if (rotationM==null) return;

        looking.set(0, 0, 0);
        rotationM.transpose();
        looking.set(1, 0, 0);
        looking.prod(rotationM);
        looking.get(lookingArray);
        PitchAzimuthCalculator.azimuth = ((Utilities.getAngle(0, 0,
lookingArray[0], lookingArray[2])  + 360 ) % 360);

        rotationM.transpose();
        looking.set(0, 1, 0);
        looking.prod(rotationM);
        looking.get(lookingArray);
        PitchAzimuthCalculator.pitch = -Utilities.getAngle(0, 0,
lookingArray[1], lookingArray[2]);
    }
}
```

Now let's take a look at the `LowPassFilter`.

LowPassFilter

A low-pass filter is an electronic filter that passes low-frequency signals, but attenuates (reduces the amplitude of) signals with frequencies higher than the cutoff frequency. The actual amount of attenuation for each frequency varies from filter to filter. It is sometimes called a *high-cut filter* or *treble cut filter* when used in audio applications.

Listing 9-51. *LowPassFilter*

```
public class LowPassFilter {

    private static final float ALPHA_DEFAULT = 0.333f;
    private static final float ALPHA_STEADY       = 0.001f;
    private static final float ALPHA_START_MOVING = 0.6f;
    private static final float ALPHA_MOVING       = 0.9f;

    private LowPassFilter() { }

    public static float[] filter(float low, float high, float[] current, float[]
previous) {
        if (current==null || previous==null)
            throw new NullPointerException("Input and prev float arrays must be
non-NULL");
        if (current.length!=previous.length)
            throw new IllegalArgumentException("Input and prev must be the same
length");

        float alpha = computeAlpha(low,high,current,previous);

        for ( int i=0; i<current.length; i++ ) {
            previous[i] = previous[i] + alpha * (current[i] - previous[i]);
        }
        return previous;
    }

    private static final float computeAlpha(float low, float high, float[]
current, float[] previous) {
        if(previous.length != 3 || current.length != 3) return ALPHA_DEFAULT;

        float x1 = current[0],
              y1 = current[1],
              z1 = current[2];

        float x2 = previous[0],
              y2 = previous[1],
              z2 = previous[2];

        float distance = (float)(Math.sqrt( Math.pow((double)(x2 - x1), 2d) +
                                            Math.pow((double)(y2 - y1), 2d) +
                                            Math.pow((double)(z2 - z1), 2d))
        );

        if(distance < low) {
            return ALPHA_STEADY;
        } else if(distance >= low || distance < high) {
            return ALPHA_START_MOVING;
        }
```

```
            return ALPHA_MOVING;
        }
    }
```

Now we'll take a look at the `Matrix` class.

Matrix

The `Matrix` class handles the functions related to Matrices like the Vector class does for Vectors. Once again, this class has been adapted from the Mixare framework.

We'll break it down as we did with the `Vector` class:

Listing 9-52. *Matrix's getters and setters, and the constructor*

```
public class Matrix {
    private static final Matrix tmp = new Matrix();

    private volatile float a1=0f, a2=0f, a3=0f;
    private volatile float b1=0f, b2=0f, b3=0f;
    private volatile float c1=0f, c2=0f, c3=0f;

    public Matrix() { }

    public synchronized float getA1() {
        return a1;
    }
    public synchronized void setA1(float a1) {
        this.a1 = a1;
    }

    public synchronized float getA2() {
        return a2;
    }
    public synchronized void setA2(float a2) {
        this.a2 = a2;
    }

    public synchronized float getA3() {
        return a3;
    }
    public synchronized void setA3(float a3) {
        this.a3 = a3;
    }

    public synchronized float getB1() {
        return b1;
    }
```

```
    public synchronized void setB1(float b1) {
        this.b1 = b1;
    }

    public synchronized float getB2() {
        return b2;
    }
    public synchronized void setB2(float b2) {
        this.b2 = b2;
    }

    public synchronized float getB3() {
        return b3;
    }
    public synchronized void setB3(float b3) {
        this.b3 = b3;
    }

    public synchronized float getC1() {
        return c1;
    }
    public synchronized void setC1(float c1) {
        this.c1 = c1;
    }

    public synchronized float getC2() {
        return c2;
    }
    public synchronized void setC2(float c2) {
        this.c2 = c2;
    }

    public synchronized float getC3() {
        return c3;
    }
    public synchronized void setC3(float c3) {
        this.c3 = c3;
    }

    public synchronized void get(float[] array) {
        if (array==null || array.length!=9)
            throw new IllegalArgumentException("get() array must be non-NULL and
size of 9");

        array[0] = this.a1;
        array[1] = this.a2;
        array[2] = this.a3;

        array[3] = this.b1;
        array[4] = this.b2;
```

```
        array[5] = this.b3;

        array[6] = this.c1;
        array[7] = this.c2;
        array[8] = this.c3;
    }

    public void set(Matrix m) {
        if (m==null) throw new NullPointerException();

        set(m.a1,m. a2, m.a3, m.b1, m.b2, m.b3, m.c1, m.c2, m.c3);
    }

    public synchronized void set(float a1, float a2, float a3, float b1, float
b2, float b3, float c1, float c2, float c3) {
        this.a1 = a1;
        this.a2 = a2;
        this.a3 = a3;

        this.b1 = b1;
        this.b2 = b2;
        this.b3 = b3;

        this.c1 = c1;
        this.c2 = c2;
        this.c3 = c3;
    }
```

The methods like `getA1()`, `getA2()`, etc. return the values of the specific part of the Matrix, while their `set()` counterparts update it. The other `get()` method populates the passed array with all nine values from the Matrix. The remaining `set()` methods set the values of the matrix to those of the supplied Matrix or to the supplied floats.

Now let's take a look at the math functions of this class:

Listing 9-53. *Matrix's Math Functions*

```
public void toIdentity() {
        set(1, 0, 0, 0, 1, 0, 0, 0, 1);
    }

public synchronized void adj() {
        float a11 = this.a1;
        float a12 = this.a2;
        float a13 = this.a3;

        float a21 = this.b1;
        float a22 = this.b2;
        float a23 = this.b3;
```

```
        float a31 = this.c1;
        float a32 = this.c2;
        float a33 = this.c3;

        this.a1 = det2x2(a22, a23, a32, a33);
        this.a2 = det2x2(a13, a12, a33, a32);
        this.a3 = det2x2(a12, a13, a22, a23);

        this.b1 = det2x2(a23, a21, a33, a31);
        this.b2 = det2x2(a11, a13, a31, a33);
        this.b3 = det2x2(a13, a11, a23, a21);

        this.c1 = det2x2(a21, a22, a31, a32);
        this.c2 = det2x2(a12, a11, a32, a31);
        this.c3 = det2x2(a11, a12, a21, a22);
    }

    public void invert() {
        float det = this.det();

        adj();
        mult(1 / det);
    }

    public synchronized void transpose() {
        float a11 = this.a1;
        float a12 = this.a2;
        float a13 = this.a3;

        float a21 = this.b1;
        float a22 = this.b2;
        float a23 = this.b3;

        float a31 = this.c1;
        float a32 = this.c2;
        float a33 = this.c3;

        this.b1 = a12;
        this.a2 = a21;
        this.b3 = a32;
        this.c2 = a23;
        this.c1 = a13;
        this.a3 = a31;

        this.a1 = a11;
        this.b2 = a22;
        this.c3 = a33;
    }
```

```
    private float det2x2(float a, float b, float c, float d) {
        return (a * d) - (b * c);
    }

    public synchronized float det() {
        return (this.a1 * this.b2 * this.c3) - (this.a1 * this.b3 * this.c2) -
(this.a2 * this.b1 * this.c3) +
        (this.a2 * this.b3 * this.c1) + (this.a3 * this.b1 * this.c2) - (this.a3
* this.b2 * this.c1);
    }

    public synchronized void mult(float c) {
        this.a1 = this.a1 * c;
        this.a2 = this.a2 * c;
        this.a3 = this.a3 * c;

        this.b1 = this.b1 * c;
        this.b2 = this.b2 * c;
        this.b3 = this.b3 * c;

        this.c1 = this.c1 * c;
        this.c2 = this.c2 * c;
        this.c3 = this.c3 * c;
    }

    public synchronized void prod(Matrix n) {
        if (n==null) throw new NullPointerException();

        tmp.set(this);
        this.a1 = (tmp.a1 * n.a1) + (tmp.a2 * n.b1) + (tmp.a3 * n.c1);
        this.a2 = (tmp.a1 * n.a2) + (tmp.a2 * n.b2) + (tmp.a3 * n.c2);
        this.a3 = (tmp.a1 * n.a3) + (tmp.a2 * n.b3) + (tmp.a3 * n.c3);

        this.b1 = (tmp.b1 * n.a1) + (tmp.b2 * n.b1) + (tmp.b3 * n.c1);
        this.b2 = (tmp.b1 * n.a2) + (tmp.b2 * n.b2) + (tmp.b3 * n.c2);
        this.b3 = (tmp.b1 * n.a3) + (tmp.b2 * n.b3) + (tmp.b3 * n.c3);

        this.c1 = (tmp.c1 * n.a1) + (tmp.c2 * n.b1) + (tmp.c3 * n.c1);
        this.c2 = (tmp.c1 * n.a2) + (tmp.c2 * n.b2) + (tmp.c3 * n.c2);
        this.c3 = (tmp.c1 * n.a3) + (tmp.c2 * n.b3) + (tmp.c3 * n.c3);
    }

    @Override
    public synchronized String toString() {
        return "(" + this.a1 + "," + this.a2 + "," + this.a3 + ")"+
               " (" + this.b1 + "," + this.b2 + "," + this.b3 + ")"+
               " (" + this.c1 + "," + this.c2 + "," + this.c3 + ")";
    }
}
```

`toIdentity()` sets the value of the matrix to 1, 0, 0, 0, 1, 0, 0, 0, 1. `adj()` finds the adjoint of the Matrix. `invert()` inverts the matrix by calling `adj()` and then dividing it by the determinant found by calling the `det()` method. The `transpose()` method transposes the matrix. The `det2x2()` method finds the determinant for the supplied values, while the `det()` method does it for the entire matrix. `mult()` multiplies every value in the matrix with the supplied float, while `prod()` multiplies the matrix with the supplied `Matrix`. `toString()` returns all the values in a human readable string format.

Now let's write the classes for the main components, namely the `Radar`, `Marker`, and `IconMarker` classes.

Components

These classes are the ones responsible for the major compononents of our app, like the Radar and the Marker. The Marker component is divided into two classes, IconMarker and Marker.

Radar

The `Radar` class is used to draw our Radar, along with all its elements like the lines and points representing the Markers.

We'll start by looking at the Global variables and the constructor:

Listing 9-54. *The Variables and Constructor of the Radar class*

```
public class Radar {
    public static final float RADIUS = 48;
    private static final int LINE_COLOR = Color.argb(150,0,0,220);
    private static final float PAD_X = 10;
    private static final float PAD_Y = 20;
    private static final int RADAR_COLOR = Color.argb(100, 0, 0, 200);
    private static final int TEXT_COLOR = Color.rgb(255,255,255);
    private static final int TEXT_SIZE = 12;

    private static ScreenPositionUtility leftRadarLine = null;
    private static ScreenPositionUtility rightRadarLine = null;
    private static PaintablePosition leftLineContainer = null;
    private static PaintablePosition rightLineContainer = null;
    private static PaintablePosition circleContainer = null;

    private static PaintableRadarPoints radarPoints = null;
    private static PaintablePosition pointsContainer = null;
```

```
    private static PaintableText paintableText = null;
    private static PaintablePosition paintedContainer = null;

    public Radar() {
        if (leftRadarLine==null) leftRadarLine = new ScreenPositionUtility();
        if (rightRadarLine==null) rightRadarLine = new ScreenPositionUtility();
    }
```

The first seven constants set the values for the colors of the Radar, its Radius, the text color, and the padding. The remaining variables are created to be `null` objects of various classes that will be initialized later. In the constructor, we check to see whether we have already created the radar lines showing the area currently being viewed. If they haven't been created, they are created to be new instances of the `ScreenPositionUtility`.

Now let's add the actual methods of the class:

Listing 9-55. *Radar's Methods*

```
public void draw(Canvas canvas) {
        if (canvas==null) throw new NullPointerException();

        PitchAzimuthCalculator.calcPitchBearing(ARData.getRotationMatrix());
        ARData.setAzimuth(PitchAzimuthCalculator.getAzimuth());
        ARData.setPitch(PitchAzimuthCalculator.getPitch());

        drawRadarCircle(canvas);
        drawRadarPoints(canvas);
        drawRadarLines(canvas);
        drawRadarText(canvas);
    }

    private void drawRadarCircle(Canvas canvas) {
        if (canvas==null) throw new NullPointerException();

        if (circleContainer==null) {
            PaintableCircle paintableCircle = new
PaintableCircle(RADAR_COLOR,RADIUS,true);
            circleContainer = new
PaintablePosition(paintableCircle,PAD_X+RADIUS,PAD_Y+RADIUS,0,1);
        }
        circleContainer.paint(canvas);
    }

    private void drawRadarPoints(Canvas canvas) {
        if (canvas==null) throw new NullPointerException();

        if (radarPoints==null) radarPoints = new PaintableRadarPoints();

        if (pointsContainer==null)
```

```
                pointsContainer = new PaintablePosition( radarPoints,
                                                         PAD_X,
                                                         PAD_Y,
                                                         -ARData.getAzimuth(),
                                                         1);
        else
                pointsContainer.set(radarPoints,
                                    PAD_X,
                                    PAD_Y,
                                    -ARData.getAzimuth(),
                                    1);

        pointsContainer.paint(canvas);
    }

    private void drawRadarLines(Canvas canvas) {
        if (canvas==null) throw new NullPointerException();

        if (leftLineContainer==null) {
            leftRadarLine.set(0, -RADIUS);
            leftRadarLine.rotate(-CameraModel.DEFAULT_VIEW_ANGLE / 2);
            leftRadarLine.add(PAD_X+RADIUS, PAD_Y+RADIUS);

            float leftX = leftRadarLine.getX()-(PAD_X+RADIUS);
            float leftY = leftRadarLine.getY()-(PAD_Y+RADIUS);
            PaintableLine leftLine = new PaintableLine(LINE_COLOR, leftX,
leftY);
            leftLineContainer = new PaintablePosition(  leftLine,
                                                        PAD_X+RADIUS,
                                                        PAD_Y+RADIUS,
                                                        0,
                                                        1);
        }
        leftLineContainer.paint(canvas);

        if (rightLineContainer==null) {
            rightRadarLine.set(0, -RADIUS);
            rightRadarLine.rotate(CameraModel.DEFAULT_VIEW_ANGLE / 2);
            rightRadarLine.add(PAD_X+RADIUS, PAD_Y+RADIUS);

            float rightX = rightRadarLine.getX()-(PAD_X+RADIUS);
            float rightY = rightRadarLine.getY()-(PAD_Y+RADIUS);
            PaintableLine rightLine = new PaintableLine(LINE_COLOR, rightX,
rightY);
            rightLineContainer = new PaintablePosition( rightLine,
                                                        PAD_X+RADIUS,
                                                        PAD_Y+RADIUS,
                                                        0,
                                                        1);
        }
```

```
        rightLineContainer.paint(canvas);
    }

    private void drawRadarText(Canvas canvas) {
        if (canvas==null) throw new NullPointerException();
        int range = (int) (ARData.getAzimuth() / (360f / 16f));
        String  dirTxt = "";
        if (range == 15 || range == 0) dirTxt = "N";
        else if (range == 1 || range == 2) dirTxt = "NE";
        else if (range == 3 || range == 4) dirTxt = "E";
        else if (range == 5 || range == 6) dirTxt = "SE";
        else if (range == 7 || range == 8) dirTxt= "S";
        else if (range == 9 || range == 10) dirTxt = "SW";
        else if (range == 11 || range == 12) dirTxt = "W";
        else if (range == 13 || range == 14) dirTxt = "NW";
        int bearing = (int) ARData.getAzimuth();
        radarText(  canvas,
                    ""+bearing+((char)176)+" "+dirTxt,
                    (PAD_X + RADIUS),
                    (PAD_Y - 5),
                    true
                );

        radarText(  canvas,
                    formatDist(ARData.getRadius() * 1000),
                    (PAD_X + RADIUS),
                    (PAD_Y + RADIUS*2 -10),
                    false
                );
    }

    private void radarText(Canvas canvas, String txt, float x, float y, boolean
bg) {
        if (canvas==null || txt==null) throw new NullPointerException();

        if (paintableText==null) paintableText = new
PaintableText(txt,TEXT_COLOR,TEXT_SIZE,bg);
        else paintableText.set(txt,TEXT_COLOR,TEXT_SIZE,bg);

        if (paintedContainer==null) paintedContainer = new
PaintablePosition(paintableText,x,y,0,1);
        else paintedContainer.set(paintableText,x,y,0,1);

        paintedContainer.paint(canvas);
    }

    private static String formatDist(float meters) {
        if (meters < 1000) {
            return ((int) meters) + "m";
        } else if (meters < 10000) {
```

```
            return formatDec(meters / 1000f, 1) + "km";
        } else {
            return ((int) (meters / 1000f)) + "km";
        }
    }

    private static String formatDec(float val, int dec) {
        int factor = (int) Math.pow(10, dec);

        int front = (int) (val);
        int back = (int) Math.abs(val * (factor) ) % factor;

        return front + "." + back;
    }
}
```

The `draw()` method starts the process by getting the pitch and azimuth and then calling the other drawing methods in the required order. `drawRadarCircle()` simply draws the base circle for the Radar. `drawRadarPoints()` draws all the points denoting Markers on the Radar circle. `drawRadarLines()` draws the two lines that show which of the markers are currently within the camera's viewing area. `drawRadarText()` calls `radarText()` format the text, before drawing it onto the Radar.

This brings us to the end of the `Radar` class. Now let's take a look at the `Marker` class.

Marker

The `Marker` class handles the majority of the coding related to the Markers we display. It calculates whether the marker should be visible on our screen and draws the image and text accordingly.

Global variables

We'll start as usual with the global variables:

Listing 9-56. *Global Variables*

```
public class Marker implements Comparable<Marker> {
    private static final DecimalFormat DECIMAL_FORMAT = new DecimalFormat("@#");

    private static final Vector symbolVector = new Vector(0, 0, 0);
    private static final Vector textVector = new Vector(0, 1, 0);

    private final Vector screenPositionVector = new Vector();
```

```
    private final Vector tmpSymbolVector = new Vector();
    private final Vector tmpVector = new Vector();
    private final Vector tmpTextVector = new Vector();
    private final float[] distanceArray = new float[1];
    private final float[] locationArray = new float[3];
    private final float[] screenPositionArray = new float[3];

    private float initialY = 0.0f;

    private volatile static CameraModel cam = null;

    private volatile PaintableBoxedText textBox = null;
    private volatile PaintablePosition textContainer = null;

    protected final float[] symbolArray = new float[3];
    protected final float[] textArray = new float[3];

    protected volatile PaintableObject gpsSymbol = null;
    protected volatile PaintablePosition symbolContainer = null;
    protected String name = null;
    protected volatile PhysicalLocationUtility physicalLocation = new
PhysicalLocationUtility();
    protected volatile double distance = 0.0;
    protected volatile boolean isOnRadar = false;
    protected volatile boolean isInView = false;
    protected final Vector symbolXyzRelativeToCameraView = new Vector();
    protected final Vector textXyzRelativeToCameraView = new Vector();
    protected final Vector locationXyzRelativeToPhysicalLocation = new Vector();
    protected int color = Color.WHITE;

    private static boolean debugTouchZone = false;
    private static PaintableBox touchBox = null;
    private static PaintablePosition touchPosition = null;

    private static boolean debugCollisionZone = false;
    private static PaintableBox collisionBox = null;
    private static PaintablePosition collisionPosition = null;
```

`DECIMAL_FORMAT` is used to format the distance we show on the Radar.

`symbolVector` and `textVector` are used to find the location of the text and the marker symbol. `symbolVector` and `textVector` are used when finding the location of the text and its accompanying symbol using the rotation matrix.

The next four vectors and three float arrays are used in positioning and drawing the marker symbol and its accompanying text.

`initialY` is the initial Y-axis position for each marker. It is set to 0 to begin with, but its value is different for each marker.

`textBox`, `textContainer`, and `cam` are instances of the `PaintableBoxedText`, `PaintablePostion`, and `CameraModel` respectively. We have not yet written `CameraModel`; we will do so after we finish all the UI pieces of the app.

`symbolArray` and `textArray` are used when drawing the symbol and text later on in this class.

`gpsSymbol` is, well, the GPS symbol. `symbolContainer` is the container for the GPS symbol. `name` is a unique identifier for each marker, set using the article title for Wikipedia articles and the username for Tweets. `physicalLocation` is the physical location of the marker (the real-world position). `distance` stores the distance from the user to the `physicalLocation` in meters. `isOnRadar` and `isInView` are used as flags to keep track of the marker's visibility. `symbolXyzRelativeToCameraView`, `textXyzRelativeToCameraView`, and `locationXyzRelativeToPhysicalLocation` are used to determine the location of the marker symbol and text relative to the camera view, and the location of the user relative to the physical location, respectively. x is up/down; y is left/right; and z is not used, but is there to complete the Vector. The color `int` is the default color of the marker, and is set to white.

`debugTouchZone` and `debugCollisionZone` are two flags we use to enable and disable debugging for both of the zones. `touchBox`, `touchPosition`, `collisionBox`, and `collisionPosition` are used to draw opaque boxes to help us debug the app.

Figure 9-1 shows the app running without `debugTouchZone` and `debugCollisionZone` set to `false`; in Figure 9.2 they are set to `true`.

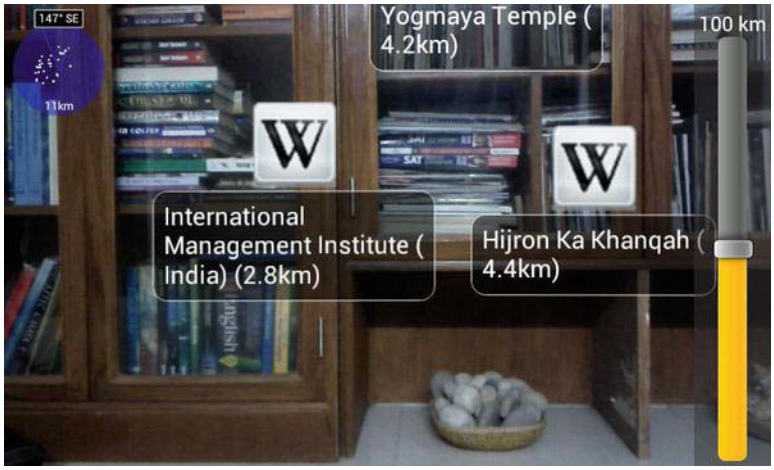

Figure 9-2. *The app running with touch and collision debugging disabled*

Figure 9-3. *The app running with touch and collision debugging enabled*

The Constructor and set() method

Now let's take a look at the constructor and `set()` method:

Listing 9-57. *The constructor and set() method*

```
        public Marker(String name, double latitude, double longitude, double
altitude, int color) {
                set(name, latitude, longitude, altitude, color);
 }
        public synchronized void set(String name, double latitude, double
longitude, double altitude, int color) {
                if (name==null) throw new NullPointerException();

  this.name               = name;
  this.physicalLocation.set(latitude,longitude,altitude);
  this.color              = color;
  this.isOnRadar              = false;
  this.isInView               = false;
  this.symbolXyzRelativeToCameraView.set(0,              0, 0);
  this.textXyzRelativeToCameraView.set(0,              0, 0);
  this.locationXyzRelativeToPhysicalLocation.set(0,              0, 0);
  this.initialY               = 0.0f;
 }
```

The constructor takes the Marker's name; its latitude, longitude, and altitude for the `PhysicalLocation`; and its color as parameters and then passes them to the `set()` method. The `set()` method sets these values to our variables described

and given in Listing 9-56. It also handles some basic initialization for the camera and the marker's position on screen.

The get() methods

Now let's take a look at the various `get()` methods for the `Marker` class:

Listing 9-58. *The get() methods*

```
public synchronized String getName(){
  return               this.name;
}

    public synchronized int getColor() {
        return this.color;
    }

    public synchronized double getDistance() {
        return this.distance;
    }

    public synchronized float getInitialY() {
        return this.initialY;
    }

    public synchronized boolean isOnRadar() {
        return this.isOnRadar;
    }

    public synchronized boolean isInView() {
        return this.isInView;
    }

    public synchronized Vector getScreenPosition() {
        symbolXyzRelativeToCameraView.get(symbolArray);
        textXyzRelativeToCameraView.get(textArray);
        float x = (symbolArray[0] + textArray[0])/2;
        float y = (symbolArray[1] + textArray[1])/2;
        float z = (symbolArray[2] + textArray[2])/2;

        if (textBox!=null) y += (textBox.getHeight()/2);

        screenPositionVector.set(x, y, z);
        return screenPositionVector;
    }

    public synchronized Vector getLocation() {
        return this.locationXyzRelativeToPhysicalLocation;
    }
```

```
    public synchronized float getHeight() {
        if (symbolContainer==null || textContainer==null) return 0f;
        return symbolContainer.getHeight()+textContainer.getHeight();
    }

    public synchronized float getWidth() {
        if (symbolContainer==null || textContainer==null) return 0f;
        float w1 = textContainer.getWidth();
        float w2 = symbolContainer.getWidth();
        return (w1>w2)?w1:w2;
    }
```

`getName()`, `getColor()`, `getDistance()`, `getLocation()`, `isInView()`, `isOnRadar()`, and `getInitialY()` simply return the values indicated by the name. `getHeight()` adds up the height of the text and the symbol image and returns it. `getWidth()` checks and returns the greater width between the text and the symbol image. `getScreenPosition()` calculates the marker's position on the screen by using the positions of the text and symbol, relative to the camera view.

The update() and populateMatrices() methods

Now let's take a look at the `update()` and `populateMatrices()` methods:

Listing 9-59. *update() and populateMatrices()*

```
public synchronized void update(Canvas canvas, float addX, float addY) {
        if (canvas==null) throw new NullPointerException();

        if (cam==null) cam = new CameraModel(canvas.getWidth(),
canvas.getHeight(), true);
        cam.set(canvas.getWidth(), canvas.getHeight(), false);
        cam.setViewAngle(CameraModel.DEFAULT_VIEW_ANGLE);
        populateMatrices(cam, addX, addY);
        updateRadar();
        updateView();
    }

        private synchronized void populateMatrices(CameraModel cam, float addX,
float addY) {
                if (cam==null) throw new NullPointerException();

  tmpSymbolVector.set(symbolVector);
                tmpSymbolVector.add(locationXyzRelativeToPhysicalLocation);
        tmpSymbolVector.prod(ARData.getRotationMatrix());

  tmpTextVector.set(textVector);
```

```
  tmpTextVector.add(locationXyzRelativeToPhysicalLocation);
  tmpTextVector.prod(ARData.getRotationMatrix());

                cam.projectPoint(tmpSymbolVector, tmpVector, addX, addY);
  symbolXyzRelativeToCameraView.set(tmpVector);
                cam.projectPoint(tmpTextVector, tmpVector, addX, addY);
  textXyzRelativeToCameraView.set(tmpVector);
 }
```

The `update()` method is used to update the views and populate the matrices. We first ensure that the `canvas` is not a `null` value, and initialize `cam` if it hasn't already been initialized. We then update the properties of `cam` to go with the `canvas` being used, and set its viewing angle. The viewing angle is defined in `CameraModel`, a class that we will be writing later on in this chapter. It then calls the `populateMatrices()` method, passing the `cam` object, and the values to be added to the X and Y position of the marker as parameters. After that, `update()` further calls `updateRadar()` and `updateView()`. In `populateMatrices()`, we find the position of the text and symbol of the marker given the rotation matrix we get from `ARData`, a class that we will be writing later on in this chapter. We then use that data to project the text and symbol onto the camera view.

The updateView() and updateRadar() Methods

Now let's take a look at the `updateView()` and `updateRadar()` methods called by `update()`.

Listing 9-60. *updateRadar() and updateView()*

```
       private synchronized void updateRadar() {
  isOnRadar               = false;

                float range = ARData.getRadius() * 1000;
                float scale = range / Radar.RADIUS;
  locationXyzRelativeToPhysicalLocation.get(locationArray);
       float x = locationArray[0] / scale;
       float y = locationArray[2] / scale; // z==y Switched on purpose
       symbolXyzRelativeToCameraView.get(symbolArray);
                if ((symbolArray[2] < -1f) &&
((x*x+y*y)<(Radar.RADIUS*Radar.RADIUS))) {
   isOnRadar                      = true;
                }
 }

    private synchronized void updateView() {
       isInView = false;

       symbolXyzRelativeToCameraView.get(symbolArray);
```

```
        float x1 = symbolArray[0] + (getWidth()/2);
        float y1 = symbolArray[1] + (getHeight()/2);
        float x2 = symbolArray[0] - (getWidth()/2);
        float y2 = symbolArray[1] - (getHeight()/2);
        if (x1>=-1 && x2<=(cam.getWidth())
            &&
            y1>=-1 && y2<=(cam.getHeight())
        ) {
            isInView = true;
        }
    }
```

`updateRadar()` is used to update the position of the marker on the radar. If the marker's location is found to be such that it should show on the radar, its `OnRadar` is updated to `true`. `updateView()` does the same thing as `updateRadar()`, except it checks to see whether the marker is currently visible.

The calcRelativePosition() and updateDistance() methods

Now let's take a look at the `calcRelativePosition()` and `updateDistance()` methods:

Listing 9-61. *calcRelativePosition() and updateDistance()*

```
public synchronized void calcRelativePosition(Location location) {
                if (location==null) throw new NullPointerException();

            updateDistance(location);

  if               (physicalLocation.getAltitude()==0.0)
physicalLocation.setAltitude(location.getAltitude());

  PhysicalLocationUtility.convLocationToVector(location,
physicalLocation, locationXyzRelativeToPhysicalLocation);
  this.initialY               = locationXyzRelativeToPhysicalLocation.getY();
  updateRadar();
    }

    private synchronized void updateDistance(Location location) {
        if (location==null) throw new NullPointerException();

        Location.distanceBetween(physicalLocation.getLatitude(),
physicalLocation.getLongitude(), location.getLatitude(),
location.getLongitude(), distanceArray);
        distance = distanceArray[0];
    }
```

In `calcRelativePosition()`, we calculate the new relative position using the location received as a parameter. We check to see whether we have a valid altitude for the marker in the `physicalLocation`; if we don't, we set it to the user's current altitude. We then use the data to create a vector, use the vector to update the `initialY` variable, and finally we call `updateRadar()` to update the radar with the new relative location. `updateDistance()` simply calculates the new distance between the physical location of the marker, and the user's location.

The handleClick(), isMarkerOnMarker(), and isPointOnMarker() Methods

Now let's take a look at how we handle clicks and see whether the marker is overlapping with another marker:

Listing 9-62. *Checking for clicks and overlaps*

```
public synchronized boolean handleClick(float x, float y) {
        if (!isOnRadar || !isInView) return false;
        return isPointOnMarker(x,y,this);
    }

    public synchronized boolean isMarkerOnMarker(Marker marker) {
        return isMarkerOnMarker(marker,true);
    }

    private synchronized boolean isMarkerOnMarker(Marker marker, boolean
reflect) {
        marker.getScreenPosition().get(screenPositionArray);
        float x = screenPositionArray[0];
        float y = screenPositionArray[1];
        boolean middleOfMarker = isPointOnMarker(x,y,this);
        if (middleOfMarker) return true;

        float halfWidth = marker.getWidth()/2;
        float halfHeight = marker.getHeight()/2;

        float x1 = x - halfWidth;
        float y1 = y - halfHeight;
        boolean upperLeftOfMarker = isPointOnMarker(x1,y1,this);
        if (upperLeftOfMarker) return true;

        float x2 = x + halfWidth;
        float y2 = y1;
        boolean upperRightOfMarker = isPointOnMarker(x2,y2,this);
        if (upperRightOfMarker) return true;

        float x3 = x1;
```

```
        float y3 = y + halfHeight;
        boolean lowerLeftOfMarker = isPointOnMarker(x3,y3,this);
        if (lowerLeftOfMarker) return true;

        float x4 = x2;
        float y4 = y3;
        boolean lowerRightOfMarker = isPointOnMarker(x4,y4,this);
        if (lowerRightOfMarker) return true;

        return (reflect)?marker.isMarkerOnMarker(this,false):false;
    }

        private synchronized boolean isPointOnMarker(float x, float y, Marker
marker) {
        marker.getScreenPosition().get(screenPositionArray);
        float myX = screenPositionArray[0];
        float myY = screenPositionArray[1];
        float adjWidth = marker.getWidth()/2;
        float adjHeight = marker.getHeight()/2;

        float x1 = myX-adjWidth;
        float y1 = myY-adjHeight;
        float x2 = myX+adjWidth;
        float y2 = myY+adjHeight;

        if (x>=x1 && x<=x2 && y>=y1 && y<=y2) return true;

        return false;
 }
```

`handleClick()` takes the X and Y points of the click as arguments. If the marker isn't on the radar and isn't in view, it returns `false`. Otherwise, it returns whatever is found by calling `isPointOnMarker()`.

The first `isMarkerOnMarker()` simply returns whatever the second `isMarkerOnMarker()` method finds out. The second `isMarkerOnMarker()` method contains all the code we use to determine whether the marker received as the parameter is overlapping with the current marker. We check for overlaps on all four corners and the center of the marker. If any of them is `true`, we can safely say that the markers are overlapping.

`isPointOnMarker()` checks to see whether the passed X and Y coordinates are located on the marker.

The draw() method

Now let's take a look at the `draw()` method:

Listing 9-63. *draw()*

```
public synchronized void draw(Canvas canvas) {
        if (canvas==null) throw new NullPointerException();

        if (!isOnRadar || !isInView) return;

        if (debugTouchZone) drawTouchZone(canvas);
        if (debugCollisionZone) drawCollisionZone(canvas);
        drawIcon(canvas);
        drawText(canvas);
    }
```

The `draw()` method is very simple. It determines whether the marker should be shown. If it is to be shown, it draws it and draws the debug boxes if required. That's all it does.

The drawTouchZone(), drawCollisionZone(), drawIcon(), and drawText() Methods

The main work for drawing is done in the `drawTouchZone()`, `drawCollisionZone()`, `drawIcon()`, and `drawText()` methods, which we will take a look at now:

Listing 9-64. *drawTouchZone(), drawCollisionZone(), drawIcon(), and drawText()*

```
protected synchronized void drawCollisionZone(Canvas canvas) {
        if (canvas==null) throw new NullPointerException();

        getScreenPosition().get(screenPositionArray);
        float x = screenPositionArray[0];
        float y = screenPositionArray[1];

        float width = getWidth();
        float height = getHeight();
        float halfWidth = width/2;
        float halfHeight = height/2;

        float x1 = x - halfWidth;
        float y1 = y - halfHeight;

        float x2 = x + halfWidth;
        float y2 = y1;

        float x3 = x1;
        float y3 = y + halfHeight;
```

```
        float x4 = x2;
        float y4 = y3;

        Log.w("collisionBox", "ul (x="+x1+" y="+y1+")");
        Log.w("collisionBox", "ur (x="+x2+" y="+y2+")");
        Log.w("collisionBox", "ll (x="+x3+" y="+y3+")");
        Log.w("collisionBox", "lr (x="+x4+" y="+y4+")");

        if (collisionBox==null) collisionBox = new
PaintableBox(width,height,Color.WHITE,Color.RED);
        else collisionBox.set(width,height);

        float currentAngle = Utilities.getAngle(symbolArray[0], symbolArray[1],
textArray[0], textArray[1])+90;

        if (collisionPosition==null) collisionPosition = new
PaintablePosition(collisionBox, x1, y1, currentAngle, 1);
        else collisionPosition.set(collisionBox, x1, y1, currentAngle, 1);
        collisionPosition.paint(canvas);
    }

    protected synchronized void drawTouchZone(Canvas canvas) {
        if (canvas==null) throw new NullPointerException();

        if (gpsSymbol==null) return;

        symbolXyzRelativeToCameraView.get(symbolArray);
        textXyzRelativeToCameraView.get(textArray);
        float x1 = symbolArray[0];
        float y1 = symbolArray[1];
        float x2 = textArray[0];
        float y2 = textArray[1];
        float width = getWidth();
        float height = getHeight();
        float adjX = (x1 + x2)/2;
        float adjY = (y1 + y2)/2;
        float currentAngle = Utilities.getAngle(symbolArray[0], symbolArray[1],
textArray[0], textArray[1])+90;
        adjX -= (width/2);
        adjY -= (gpsSymbol.getHeight()/2);

        Log.w("touchBox", "ul (x="+(adjX)+" y="+(adjY)+")");
        Log.w("touchBox", "ur (x="+(adjX+width)+" y="+(adjY)+")");
        Log.w("touchBox", "ll (x="+(adjX)+" y="+(adjY+height)+")");
        Log.w("touchBox", "lr (x="+(adjX+width)+" y="+(adjY+height)+")");

        if (touchBox==null) touchBox = new
PaintableBox(width,height,Color.WHITE,Color.GREEN);
        else touchBox.set(width,height);
```

```
        if (touchPosition==null) touchPosition = new PaintablePosition(touchBox,
adjX, adjY, currentAngle, 1);
        else touchPosition.set(touchBox, adjX, adjY, currentAngle, 1);
        touchPosition.paint(canvas);
    }

    protected synchronized void drawIcon(Canvas canvas) {
        if (canvas==null) throw new NullPointerException();

        if (gpsSymbol==null) gpsSymbol = new PaintableGps(36, 36, true,
getColor());

        textXyzRelativeToCameraView.get(textArray);
        symbolXyzRelativeToCameraView.get(symbolArray);

        float currentAngle = Utilities.getAngle(symbolArray[0], symbolArray[1],
textArray[0], textArray[1]);
        float angle = currentAngle + 90;

        if (symbolContainer==null) symbolContainer = new
PaintablePosition(gpsSymbol, symbolArray[0], symbolArray[1], angle, 1);
        else symbolContainer.set(gpsSymbol, symbolArray[0], symbolArray[1],
angle, 1);

        symbolContainer.paint(canvas);
    }

    protected synchronized void drawText(Canvas canvas) {
                if (canvas==null) throw new NullPointerException();

            String textStr = null;
            if (distance<1000.0) {
                textStr = name + " ("+ DECIMAL_FORMAT.format(distance) + "m)";
            } else {
                double d=distance/1000.0;
                textStr = name + " (" + DECIMAL_FORMAT.format(d) + "km)";
            }

            textXyzRelativeToCameraView.get(textArray);
            symbolXyzRelativeToCameraView.get(symbolArray);

            float maxHeight = Math.round(canvas.getHeight() / 10f) + 1;
            if (textBox==null) textBox = new PaintableBoxedText(textStr,
Math.round(maxHeight / 2f) + 1, 300);
            else textBox.set(textStr, Math.round(maxHeight / 2f) + 1, 300);

            float currentAngle = Utilities.getAngle(symbolArray[0],
symbolArray[1], textArray[0], textArray[1]);
        float angle = currentAngle + 90;
```

```
            float x = textArray[0] - (textBox.getWidth() / 2);
            float y = textArray[1] + maxHeight;

            if (textContainer==null) textContainer = new
PaintablePosition(textBox, x, y, angle, 1);
            else textContainer.set(textBox, x, y, angle, 1);
            textContainer.paint(canvas);
 }
```

The `drawCollisionZone()` method draws the collision zone between two markers if there is one to be drawn. The `drawTouchZone()` draws a red rectangle over the area we listen to for touches on the marker. The `drawIcon()` method draws the icon, and the `drawText()` method draws the related text.

The compareTo() and equals() Methods

Now let's take a look at the final two methods, `compareTo()` and `equals()`:

Listing 9-65. *compareTo() and equals()*

```
public synchronized int compareTo(Marker another) {
        if (another==null) throw new NullPointerException();

        return name.compareTo(another.getName());
    }

    @Override
    public synchronized boolean equals(Object marker) {
        if(marker==null || name==null) throw new NullPointerException();

        return name.equals(((Marker)marker).getName());
    }
}
```

`compareTo()` compares the names of the two markers using the standard Java String functions. `equals()` uses the standard Java String functions to check the name of one marker against another.

This brings us to the end of the `Marker.java` file. Now let's take a look at its only subclass, `IconMarker.java`.

IconMarker.java

`IconMarker` draws a bitmap as an icon for a marker, instead of leaving it at the default. It is an extension of `Marker.java`.

Listing 9-66. *IconMarker*

```
public class IconMarker extends Marker {
    private Bitmap bitmap = null;

    public IconMarker(String name, double latitude, double longitude, double
altitude, int color, Bitmap bitmap) {
        super(name, latitude, longitude, altitude, color);
        this.bitmap = bitmap;
    }

    @Override
    public void drawIcon(Canvas canvas) {
        if (canvas==null || bitmap==null) throw new NullPointerException();

        if (gpsSymbol==null) gpsSymbol = new PaintableIcon(bitmap,96,96);

        textXyzRelativeToCameraView.get(textArray);
        symbolXyzRelativeToCameraView.get(symbolArray);

        float currentAngle = Utilities.getAngle(symbolArray[0], symbolArray[1],
textArray[0], textArray[1]);
        float angle = currentAngle + 90;

        if (symbolContainer==null) symbolContainer = new
PaintablePosition(gpsSymbol, symbolArray[0], symbolArray[1], angle, 1);
        else symbolContainer.set(gpsSymbol, symbolArray[0], symbolArray[1],
angle, 1);

        symbolContainer.paint(canvas);
    }
}
```

The constructor takes all the parameters required for a `super()` call to `Marker.java`, along with an extra parameter that is the bitmap for the marker. `drawIcon()` then uses data from `Marker.java` to draw the bitmap we received in the constructor as the icon for this marker.

This brings us to the end of the UI components. Now let's take a look at `VerticalSeekBar.java`, our custom extension of the Android SeekBar.

Customized Widget

We have one customization of the standard Android widgets in our app. We have extended `SeekBar` to create `VerticalSeekBar`.

VerticalSeekBar.java

`VerticalSeekBar` is an extension of Android's `SeekBar` implementation. Our additional code allows it to work vertically instead of horizontally. The `zoomBar` used in the app is an instance of this class.

Listing 9-67. *VerticalSeekBar.java*

```
public class VerticalSeekBar extends SeekBar {

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

    public VerticalSeekBar(Context context, AttributeSet attrs, int defStyle) {
        super(context, attrs, defStyle);
    }

    public VerticalSeekBar(Context context, AttributeSet attrs) {
        super(context, attrs);
    }

    @Override
    protected void onSizeChanged(int w, int h, int oldw, int oldh) {
        super.onSizeChanged(h, w, oldh, oldw);
    }

    @Override
    protected synchronized void onMeasure(int widthMeasureSpec, int
heightMeasureSpec) {
        super.onMeasure(heightMeasureSpec, widthMeasureSpec);
        setMeasuredDimension(getMeasuredHeight(), getMeasuredWidth());
    }

    @Override
    protected void onDraw(Canvas c) {
        c.rotate(-90);
        c.translate(-getHeight(), 0);

        super.onDraw(c);
    }

    @Override
    public boolean onTouchEvent(MotionEvent event) {
        if (!isEnabled()) {
            return false;
        }

        switch (event.getAction()) {
```

```
        case MotionEvent.ACTION_DOWN:
        case MotionEvent.ACTION_MOVE:
        case MotionEvent.ACTION_UP:
            setProgress(getMax() - (int) (getMax() * event.getY() /
getHeight()));
            onSizeChanged(getWidth(), getHeight(), 0, 0);
            break;

        case MotionEvent.ACTION_CANCEL:
            break;
        }
        return true;
    }
}
```

Our three constructors are used to tie it to the parent `SeekBar` class. `onSizeChanged()` also ties back to the `SeekBar` class. `onMeasure()` does a `super()` call, and also sets the measured height and width using the methods provided by Android's `View` class. The actual modification is done in `onDraw()`, when we rotate the canvas by 90 degrees before passing it to `SeekBar`, so that the drawing is done vertically, not horizontally. In `onTouchEvent()`, we call `setProgress()` and `onSizeChanged()`to allow for our rotation of the `SeekBar` to work properly.

Now let's take a look at the three classes required to control the camera.

Controlling the Camera

As with any AR app, we must use the camera in this one as well. Due to the nature of this app, the camera control has been put in three classes, which we will go through now.

CameraSurface.java

The first class we will look at is `CameraSurface`. This class handles all the Camera's `SurfaceView`–related code:

Listing 9-68. *Variables and Constructor*

```
public class CameraSurface extends SurfaceView implements SurfaceHolder.Callback
{
    private static SurfaceHolder holder = null;
    private static Camera camera = null;

    public CameraSurface(Context context) {
```

```
        super(context);

        try {
            holder = getHolder();
            holder.addCallback(this);
            holder.setType(SurfaceHolder.SURFACE_TYPE_PUSH_BUFFERS);
        } catch (Exception ex) {
                ex.printStackTrace();
        }
    }
```

We declare the `SurfaceHolder` and `camera` right at the beginning. We then use the constructor to initialize the `SurfaceHolder` and set its type to `SURFACE_TYPE_PUSH_BUFFERS`, which allows it to receive data from the camera.

Now let's take a look at the `surfaceCreated()` method:

Listing 9-69. *surfaceCreated()*

```
public void surfaceCreated(SurfaceHolder holder) {
        try {
            if (camera != null) {
                try {
                    camera.stopPreview();
                } catch (Exception ex) {
                        ex.printStackTrace();
                }
                try {
                    camera.release();
                } catch (Exception ex) {
                        ex.printStackTrace();
                }
                camera = null;
            }

            camera = Camera.open();
            camera.setPreviewDisplay(holder);
        } catch (Exception ex) {
            try {
                if (camera != null) {
                    try {
                        camera.stopPreview();
                    } catch (Exception ex1) {
                        ex.printStackTrace();
                    }
                    try {
                        camera.release();
                    } catch (Exception ex2) {
                        ex.printStackTrace();
                    }
```

```
                camera = null;
            }
        } catch (Exception ex3) {
            ex.printStackTrace();
        }
    }
}
```

We use `surfaceCreated()` to create the `camera` object and to release it as well if it already exists or if we encounter a problem.

Now let's move on to `surfaceDestroyed()`.

Listing 9-70. *surfaceDestroyed()*

```
public void surfaceDestroyed(SurfaceHolder holder) {
        try {
            if (camera != null) {
                try {
                    camera.stopPreview();
                } catch (Exception ex) {
                        ex.printStackTrace();
                }
                try {
                    camera.release();
                } catch (Exception ex) {
                        ex.printStackTrace();
                }
                camera = null;
            }
        } catch (Exception ex) {
                ex.printStackTrace();
        }
    }
```

This is simple code, pretty much the same as our other example apps. We stop using the camera and release it so that our own or another third-party or system app can access it.

Without further ado, let's take a look at the last method of this class, `surfaceChanged()`:

Listing 9-71. *surfaceChanged()*

```
public void surfaceChanged(SurfaceHolder holder, int format, int w, int h) {
        try {
            Camera.Parameters parameters = camera.getParameters();
            try {
                List<Camera.Size> supportedSizes = null;
```

```
                supportedSizes =
CameraCompatibility.getSupportedPreviewSizes(parameters);

                float ff = (float)w/h;

                float bff = 0;
                int bestw = 0;
                int besth = 0;
                Iterator<Camera.Size> itr = supportedSizes.iterator();

                while(itr.hasNext()) {
                    Camera.Size element = itr.next();
                    float cff = (float)element.width/element.height;

                    if ((ff-cff <= ff-bff) && (element.width <= w) &&
(element.width >= bestw)) {
                        bff=cff;
                        bestw = element.width;
                        besth = element.height;
                    }
                }

                if ((bestw == 0) || (besth == 0)){
                    bestw = 480;
                    besth = 320;
                }
                parameters.setPreviewSize(bestw, besth);
            } catch (Exception ex) {
                parameters.setPreviewSize(480 , 320);
            }

            camera.setParameters(parameters);
            camera.startPreview();
        } catch (Exception ex) {
                ex.printStackTrace();
        }
    }
}
```

We use `surfaceChanged()` to calculate the preview size that gives us an aspect ratio (form factor, stored in `bff`) that is closest to the device's screen's aspect ratio (`ff`). We also make sure that the preview size is not greater than that of the screen, as some phones like the HTC Hero report sizes that are bigger and crash when an app uses them. We also have an `if` statement at the end to guard against a 0 value for the width and the height, as might occur on some Samsung phones.

Now let's move on to the remaining two classes: `CameraCompatibility` and `CameraModel`.

CameraCompatibility

`CameraCompatibility` allows our app to maintain compatibility with all versions of Android and get around limitations of the older versions' APIs. It is adapted from the Mixare project, like the `Vector` class.

Listing 9-72. *CameraCompatibility*

```
public class CameraCompatibility {
        private static Method getSupportedPreviewSizes = null;
        private static Method mDefaultDisplay_getRotation = null;

 static        {
  initCompatibility();
 };

        private static void initCompatibility() {
  try               {
   getSupportedPreviewSizes                        =
Camera.Parameters.class.getMethod("getSupportedPreviewSizes", new Class[] { } );
   mDefaultDisplay_getRotation                      =
Display.class.getMethod("getRotation", new Class[] { } );
                } catch (NoSuchMethodException nsme) {
                }
 }

        public static int getRotation(Activity activity) {
             int result = 1;
             try {
             Display display = ((WindowManager)
activity.getSystemService(Context.WINDOW_SERVICE)).getDefaultDisplay();
             Object retObj = mDefaultDisplay_getRotation.invoke(display);
             if(retObj != null) result = (Integer) retObj;
             } catch (Exception ex) {
                 ex.printStackTrace();
             }
             return result;
 }

        public static List<Camera.Size>
getSupportedPreviewSizes(Camera.Parameters params) {
                List<Camera.Size> retList = null;

  try               {
   Object                        retObj = getSupportedPreviewSizes.invoke(params);
                        if (retObj != null) {
    retList                                 = (List<Camera.Size>)retObj;
                        }
                } catch (InvocationTargetException ite) {
```

```
   Throwable                       cause = ite.getCause();
                        if (cause instanceof RuntimeException) {
    throw                               (RuntimeException) cause;
                        } else if (cause instanceof Error) {
    throw                               (Error) cause;
   }                      else {
    throw                               new RuntimeException(ite);
                        }
                } catch (IllegalAccessException ie) {
   ie.printStackTrace();
                }
  return                retList;
 }
}
```

`initCompatibility()` fails on devices lower than Android 2.0, and this allows us to gracefully set the camera to the default preview size of 480 by 320. `getRotation()` allows us to retrieve the rotation of the device. `getSupportedPreviewSizes()` returns a list of the preview sizes available on the device.

Now let's take a look at `CameraModel`, the last of the camera classes, and second-to-last class of this app.

CameraModel

`CameraModel` represents the camera and its view, and also allows us to project points. It is another class that has been adapted from Mixare.

Listing 9-73. *CameraModel*

```
public class CameraModel {
    private static final float[] tmp1 = new float[3];
    private static final float[] tmp2 = new float[3];

        private int width = 0;
        private int height = 0;
        private float distance = 0F;

        public static final float DEFAULT_VIEW_ANGLE = (float)
Math.toRadians(45);

        public CameraModel(int width, int height, boolean init) {
  set(width,               height, init);
 }

        public void set(int width, int height, boolean init) {
  this.width               = width;
```

```
 this.height               = height;
 }

        public int getWidth() {
            return width;
 }

    public int getHeight() {
        return height;
    }

        public void setViewAngle(float viewAngle) {
                this.distance = (this.width / 2) / (float) Math.tan(viewAngle /
2);
 }

        public void projectPoint(Vector orgPoint, Vector prjPoint, float addX,
float addY) {
            orgPoint.get(tmp1);
            tmp2[0]=(distance * tmp1[0] / -tmp1[2]);
            tmp2[1]=(distance * tmp1[1] / -tmp1[2]);
            tmp2[2]=(tmp1[2]);
            tmp2[0]=(tmp2[0] + addX + width / 2);
            tmp2[1]=(-tmp2[1] + addY + height / 2);
            prjPoint.set(tmp2);
 }
}
```

The constructor sets up the width and height for the class. `getWidth()` and `getHeight()` return the width and height, respectively. `setViewAngle()` updates the distance with a new viewing angle. `projectPoint()` projects a point using the origin vector, the project vector, and the additions to the X and Ycoordinates.

The Global Class

Now let's take a look at our last class, `ARData`.

ARData.java

`ARData` serves as a global control and storage class; it stores the data that is used throughout the app and is essential for its running. It makes it simpler for us to store all this data in one place, instead of having bits and pieces of it scattered all over.

Listing 9-74. *ARData's Global Variables*

```
public abstract class ARData {
    private static final String TAG = "ARData";
        private static final Map<String,Marker> markerList = new
ConcurrentHashMap<String,Marker>();
    private static final List<Marker> cache = new
CopyOnWriteArrayList<Marker>();
    private static final AtomicBoolean dirty = new AtomicBoolean(false);
    private static final float[] locationArray = new float[3];

    public static final Location hardFix = new Location("ATL");
    static {
        hardFix.setLatitude(0);
        hardFix.setLongitude(0);
        hardFix.setAltitude(1);
    }

    private static final Object radiusLock = new Object();
    private static float radius = new Float(20);
    private static String zoomLevel = new String();
    private static final Object zoomProgressLock = new Object();
    private static int zoomProgress = 0;
    private static Location currentLocation = hardFix;
    private static Matrix rotationMatrix = new Matrix();
    private static final Object azimuthLock = new Object();
    private static float azimuth = 0;
    private static final Object pitchLock = new Object();
    private static float pitch = 0;
    private static final Object rollLock = new Object();
    private static float roll = 0;
```

`TAG` is a string used when showing messages in the LogCat. `markerList` is a Hashmap of the markers and their names. `cache` is, well, a cache. `dirty` is used to tell whether the state is dirty or not. `locationArray` is an array of the location data. `hardFix` is a default location, the same as the ATL marker we have. `radius` is that radius of the radar; `zoomProgress` is the progress of the zoom in our app. `pitch`, `azimuth`, and `roll` hold the pitch, azimuth, and roll values, respectively. The previous variables with `Lock` added to the name are the lock objects for the synchronized blocks for those variables. `zoomLevel` is the level of zoom currently there. `currentLocation` stores the current location, set by default to `hardFix`. Finally, `rotationMatrix` stores the rotation matrix.

Now let's take a look at the various getter and setter methods of this class:

Listing 9-75. *The getters and setters for ARData*

```
public static void setZoomLevel(String zoomLevel) {
```

```
    if (zoomLevel==null) throw new NullPointerException();

    synchronized (ARData.zoomLevel) {
        ARData.zoomLevel = zoomLevel;
    }
}

public static void setZoomProgress(int zoomProgress) {
    synchronized (ARData.zoomProgressLock) {
        if (ARData.zoomProgress != zoomProgress) {
            ARData.zoomProgress = zoomProgress;
            if (dirty.compareAndSet(false, true)) {
                Log.v(TAG, "Setting DIRTY flag!");
                cache.clear();
            }
        }
    }
}

public static void setRadius(float radius) {
    synchronized (ARData.radiusLock) {
        ARData.radius = radius;
    }
}

public static float getRadius() {
    synchronized (ARData.radiusLock) {
        return ARData.radius;
    }
}

public static void setCurrentLocation(Location currentLocation) {
    if (currentLocation==null) throw new NullPointerException();

    Log.d(TAG, "current location. location="+currentLocation.toString());
    synchronized (currentLocation) {
        ARData.currentLocation = currentLocation;
    }
    onLocationChanged(currentLocation);
}

public static Location getCurrentLocation() {
    synchronized (ARData.currentLocation) {
        return ARData.currentLocation;
    }
}

public static void setRotationMatrix(Matrix rotationMatrix) {
    synchronized (ARData.rotationMatrix) {
        ARData.rotationMatrix = rotationMatrix;
```

```
            }
        }

        public static Matrix getRotationMatrix() {
            synchronized (ARData.rotationMatrix) {
                return rotationMatrix;
            }
        }

        public static List<Marker> getMarkers() {
            if (dirty.compareAndSet(true, false)) {
                Log.v(TAG, "DIRTY flag found, resetting all marker heights to
zero.");
                for(Marker ma : markerList.values()) {
                    ma.getLocation().get(locationArray);
                    locationArray[1]=ma.getInitialY();
                    ma.getLocation().set(locationArray);
                }

                Log.v(TAG, "Populating the cache.");
                List<Marker> copy = new ArrayList<Marker>();
                copy.addAll(markerList.values());
                Collections.sort(copy,comparator);
                cache.clear();
                cache.addAll(copy);
            }
            return Collections.unmodifiableList(cache);
        }

        public static void setAzimuth(float azimuth) {
            synchronized (azimuthLock) {
                ARData.azimuth = azimuth;
            }
        }

        public static float getAzimuth() {
            synchronized (azimuthLock) {
                return ARData.azimuth;
            }
        }

        public static void setPitch(float pitch) {
            synchronized (pitchLock) {
                ARData.pitch = pitch;
            }
        }

        public static float getPitch() {
            synchronized (pitchLock) {
                return ARData.pitch;
```

```
        }
    }

    public static void setRoll(float roll) {
        synchronized (rollLock) {
            ARData.roll = roll;
        }
    }

    public static float getRoll() {
        synchronized (rollLock) {
            return ARData.roll;
        }
    }
```

All the methods simply set or get the variable mentioned in their name, using a synchronized block to ensure that the data isn't modified by two different parts of the app at once. In the `getMarkers()` methods, we iterate over the markers to return all them. Now let's take a look at the last few methods of this class.

Listing 9-76. *addMarkers(), comparator, and onLocationChanged()*

```
private static final Comparator<Marker> comparator = new Comparator<Marker>() {
        public int compare(Marker arg0, Marker arg1) {
            return Double.compare(arg0.getDistance(),arg1.getDistance());
        }
    };

    public static void addMarkers(Collection<Marker> markers) {
        if (markers==null) throw new NullPointerException();

        if (markers.size()<=0) return;

        Log.d(TAG, "New markers, updating markers. new
markers="+markers.toString());
        for(Marker marker : markers) {
            if (!markerList.containsKey(marker.getName())) {
                marker.calcRelativePosition(ARData.getCurrentLocation());
                markerList.put(marker.getName(),marker);
            }
        }

        if (dirty.compareAndSet(false, true)) {
            Log.v(TAG, "Setting DIRTY flag!");
            cache.clear();
        }
    }

    private static void onLocationChanged(Location location) {
```

```
        Log.d(TAG, "New location, updating markers.
location="+location.toString());
        for(Marker ma: markerList.values()) {
            ma.calcRelativePosition(location);
        }

        if (dirty.compareAndSet(false, true)) {
            Log.v(TAG, "Setting DIRTY flag!");
            cache.clear();
        }
    }
}
```

`comparator` is used to compare the distance of one marker to another. `addMarkers()` is used to add new markers from the passed collection. `onLocationChanged()` is used to update the relative positions of the markers with respect to the new location.

AndroidManifest.xml

Finally, we must create the Android Manifest as follows:

Listing 9-77. *AndroidManifest.xml*

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.paar.ch9"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk android:minSdkVersion="7" />
 <uses-permission        android:name="android.permission.CAMERA"/>
 <uses-permission        android:name="android.permission.INTERNET"/>
 <uses-permission
android:name="android.permission.ACCESS_FINE_LOCATION"/>
 <uses-permission
android:name="android.permission.ACCESS_CORSE_LOCATION"/>
 <uses-permission        android:name="android.permission.WAKE_LOCK"/>

    <application
        android:icon="@drawable/ic_launcher"
                        android:label="@string/app_name"

android:theme="@android:style/Theme.NoTitleBar.Fullscreen">

        <activity android:name=".MainActivity"
                  android:label="@string/app_name"
                  android:screenOrientation="landscape">
```

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

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

Running the App

There are a few things you should keep in mind when running the app.

If there is no Internet connection, no data will turn up, and hence no markers will be shown.

There are some places where no data will be found for tweets and Wikipedia articles. This is more likely for tweets. Also, at certain times of the night, tweets might be few in number.

Some of the marker may be higher than others. This is due to the collisions avoidance and because of the altitude property in the locations.

Figures 9.4 and 9.5 show the app in action. Debugging has been disabled.

Figure 9-4. *App showing a marker*

Figure 9-5. *Several markers converge due to the compass going haywire next to a metallic object.*

Summary

This chapter covered the process and provided code for the creation of an AR browser, one of the most popular kind of AR apps. And with Google Goggles coming up, it will get even more popular among people who want to use something like the goggles without actually owing a set. Be sure to download the source code for this chapter from this book's page on Apress or from the GitHub repository. You can reach me directly at `raghavsood@appaholics.in` or via Twitter at `@Appaholics16`.