

28. Implementing Custom Gesture and Pinch Recognition on Android

The previous chapter looked at the steps involved in detecting what are referred to as “common gestures” from within an Android application. In practice, however, a gesture can conceivably involve just about any sequence of touch motions on the display of an Android device. In recognition of this fact, the Android SDK allows custom gestures of just about any nature to be defined by the application developer and used to trigger events when performed by the user. This is a multistage process, the details of which are the topic of this chapter.

28.1 The Android Gesture Builder Application

The Android SDK allows developers to design custom gestures which are then stored in a gesture file bundled with an Android application package. These custom gesture files are most easily created using the *Gesture Builder* application which is bundled with the samples package supplied as part of the Android SDK. The creation of a gestures file involves launching the Gesture Builder application, either on a physical device or emulator, and “drawing” the gestures that will need to be detected by the application. Once the gestures have been designed, the file containing the gesture data can be pulled off the SD card of the device or emulator and added to the application project. Within the application code, the file is then loaded into an instance of the *GestureLibrary* class where it can be used to search for matches to any gestures performed by the user on the device display.

28.2 The GestureOverlayView Class

In order to facilitate the detection of gestures within an application, the Android SDK provides the *GestureOverlayView* class. This is a transparent view that can be placed over other views in the user interface for the sole purpose of detecting gestures.

28.3 Detecting Gestures

Gestures are detected by loading the gestures file created using the Gesture Builder app and then registering a *GesturePerformedListener* event listener on

an instance of the `GestureOverlayView` class. The enclosing class is then declared to implement both the *OnGesturePerformedListener* interface and the corresponding *onGesturePerformed* callback method required by that interface. In the event that a gesture is detected by the listener, a call to the *onGesturePerformed* callback method is triggered by the Android runtime system.

28.4 Identifying Specific Gestures

When a gesture is detected, the *onGesturePerformed* callback method is called and passed as arguments a reference to the `GestureOverlayView` object on which the gesture was detected, together with a `Gesture` object containing information about the gesture.

With access to the `Gesture` object, the `GestureLibrary` can then be used to compare the detected gesture to those contained in the gestures file previously loaded into the application. The `GestureLibrary` reports the probability that the gesture performed by the user matches an entry in the gestures file by calculating a *prediction score* for each gesture. A prediction score of 1.0 or greater is generally accepted to be a good match between a gesture stored in the file and that performed by the user on the device display.

28.5 Building and Running the Gesture Builder Application

The Gesture Builder application is bundled by default with the AVD emulator profile for most versions of the SDK. It is not, however, pre-installed on most physical Android devices. If the utility is pre-installed, it will be listed along with the other apps installed in the device or AVD instance. In the event that it is not installed, the source code for the utility is included with the sample code provided with this book. If you have not already done so, download this now using the following link:

<http://www.ebookfrenzy.com/retail/androidstudio30/index.php>

The source code for the Gesture Builder application is located within this archive in a folder named *GestureBuilder*.

The `GestureBuilder` project is based on Android 5.0.1 (API 21) so use the SDK Manager tool once again to ensure that this version of the Android SDK is installed before proceeding.

From the Android Studio welcome screen select the *Import project* option. Alternatively, from the Android Studio main window for an existing project, select the *File -> New -> Import Project...* menu option and, within the resulting dialog, navigate to and select the GestureBuilder folder within the samples directory and click on OK. At this point, Android Studio will import the project into the designated folder and convert it to match the Android Studio project file and build structure.

Once imported, install and run the GestureBuilder utility on an Android device attached to the development system.

28.6 Creating a Gestures File

Once the Gesture Builder application has loaded, it should indicate that no gestures have yet been created. To create a new gesture, click on the *Add gesture* button located at the bottom of the device screen, enter the name *Circle Gesture* into the *Name* text box and then “draw” a gesture using a circular motion on the screen as illustrated in [Figure 28-1](#). Assuming that the gesture appears as required (represented by the yellow line on the device screen), click on the *Done* button to add the gesture to the gestures file:

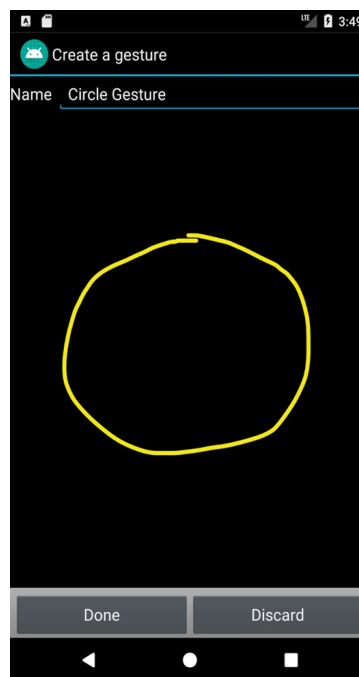


Figure 28-1

After the gesture has been saved, the Gesture Builder app will display a list of currently defined gestures, which, at this point, will consist solely of the new

Circle Gesture.

Repeat the gesture creation process to add a further gesture to the file. This should involve a two-stroke gesture creating an X on the screen named *X Gesture*. When creating gestures involving multiple strokes, be sure to allow as little time as possible between each stroke so that the builder knows that the strokes are part of the same gesture. Once this gesture has been added, the list within the Gesture Builder application should resemble that outlined in [Figure 28-2](#):

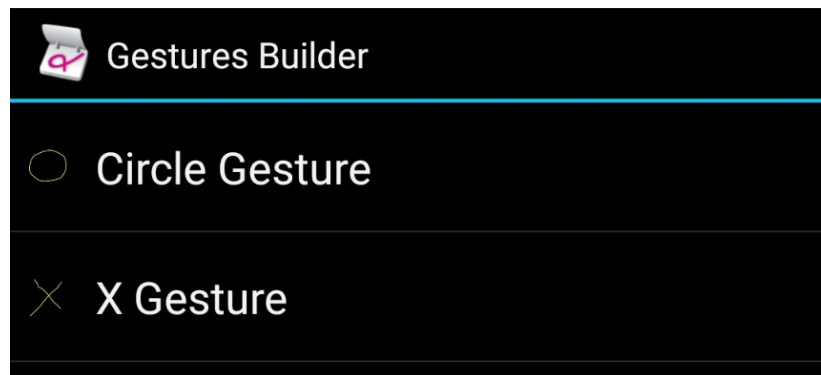


Figure 28-2

28.7 Creating the Example Project

Create a new project in Android Studio, entering *CustomGestures* into the Application name field and *ebookfrenzy.com* as the Company Domain setting before clicking on the *Next* button.

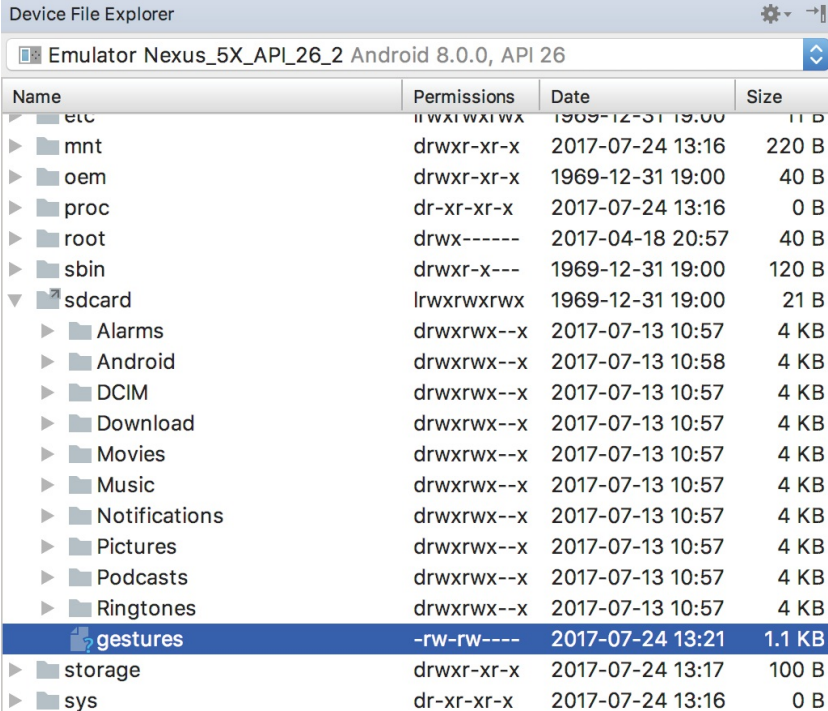
On the form factors screen, enable the *Phone and Tablet* option and set the minimum SDK setting to API 14: Android 4.0 (IceCreamSandwich). Continue to proceed through the screens, requesting the creation of an Empty Activity named *CustomGesturesActivity* with a corresponding layout file named *activity_custom_gestures*.

Click on the *Finish* button to initiate the project creation process.

28.8 Extracting the Gestures File from the SD Card

As each gesture was created within the Gesture Builder application, it was added to a file named *gestures* located on the SD Card of the emulator or device on which the app was running. Before this file can be added to an Android Studio project, however, it must first be pulled off the SD Card and saved to the local file system. This is most easily achieved by using the

Android Studio Device File Explorer tool window. Display this tool using the *View -> Tool Windows -> Device File Explorer* menu option. Once displayed, select the device on which the gesture file was created from the dropdown menu, then navigate through the filesystem to the */sdcard* folder:



Name	Permissions	Date	Size
etc	lrwxrwxrwx	1969-12-31 19:00	11 B
mnt	drwxr-xr-x	2017-07-24 13:16	220 B
oem	drwxr-xr-x	1969-12-31 19:00	40 B
proc	dr-xr-xr-x	2017-07-24 13:16	0 B
root	drwx-----	2017-04-18 20:57	40 B
sbin	drwxr-x---	1969-12-31 19:00	120 B
sdcard	lrwxrwxrwx	1969-12-31 19:00	21 B
Alarms	drwxrwx--x	2017-07-13 10:57	4 KB
Android	drwxrwx--x	2017-07-13 10:58	4 KB
DCIM	drwxrwx--x	2017-07-13 10:57	4 KB
Download	drwxrwx--x	2017-07-13 10:57	4 KB
Movies	drwxrwx--x	2017-07-13 10:57	4 KB
Music	drwxrwx--x	2017-07-13 10:57	4 KB
Notifications	drwxrwx--x	2017-07-13 10:57	4 KB
Pictures	drwxrwx--x	2017-07-13 10:57	4 KB
Podcasts	drwxrwx--x	2017-07-13 10:57	4 KB
Ringtones	drwxrwx--x	2017-07-13 10:57	4 KB
gestures	-rw-rw----	2017-07-24 13:21	1.1 KB
storage	drwxr-xr-x	2017-07-24 13:17	100 B
sys	dr-xr-xr-x	2017-07-24 13:16	0 B

Figure 28-3

Locate the *gestures* file in this folder, right click on it and select the *Save as...* menu and save the file to a temporary location.

Once the gestures file has been created and pulled off the SD Card, it is ready to be added to an Android Studio project as a resource file.

28.9 Adding the Gestures File to the Project

Within the Android Studio Project tool window, locate and right-click on the *res* folder (located under *app*) and select *New -> Directory* from the resulting menu. In the New Directory dialog, enter *raw* as the folder name and click on the *OK* button. Using the appropriate file explorer utility for your operating system type, locate the *gestures* file previously pulled from the SD Card and copy and paste it into the new *raw* folder in the Project tool window.

28.10 Designing the User Interface

This example application calls for a very simple user interface consisting of a

LinearLayout view with a GestureOverlayView layered on top of it to intercept any gestures performed by the user. Locate the *app -> res -> layout -> activity_custom_gestures.xml* file and double-click on it to load it into the Layout Editor tool.

Once loaded, switch to Text mode and modify the XML so that it reads as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    xmlns:android="http://schemas.android.com/apk/res/android">

    <android.gesture.GestureOverlayView
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:id="@+id/gOverlay"
        android:layout_gravity="center_horizontal">

        </android.gesture.GestureOverlayView>
</LinearLayout>
```

28.1 Loading the Gestures File

Now that the gestures file has been added to the project, the next step is to write some code so that the file is loaded when the activity starts up. For the purposes of this project, the code to achieve this will be added to the *CustomGesturesActivity* class located in the *CustomGesturesActivity.java* source file as follows:

```
package com.ebookfrenzy.customgestures;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.gesture.GestureLibraries;
import android.gesture.GestureLibrary;
import android.gesture.GestureOverlayView;
import android.gesture.GestureOverlayView.OnGesturePerformedListener;

public class CustomGesturesActivity extends AppCompatActivity
    implements OnGesturePerformedListener {
```

```

private GestureLibrary gLibrary;

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_custom_gestures);

    gestureSetup();
}

private void gestureSetup() {
    gLibrary =
        GestureLibraries.fromRawResource(this,
            R.raw.gestures);
    if (!gLibrary.load()) {
        finish();
    }
}
.
.
.
}

```

In addition to some necessary import directives, the above code also creates a *GestureLibrary* instance named *gLibrary* and then loads into it the contents of the gestures file located in the *raw* resources folder. The activity class has also been modified to implement the *OnGesturePerformedListener* interface, which requires the implementation of the *onGesturePerformed* callback method (which will be created in a later section of this chapter).

28.12 Registering the Event Listener

In order for the activity to receive notification that the user has performed a gesture on the screen, it is necessary to register the *OnGesturePerformedListener* event listener on the *gLayout* view, a reference to which can be obtained using the *findViewById* method as outlined in the following code fragment:

```

private void gestureSetup() {
    gLibrary =
        GestureLibraries.fromRawResource(this,
            R.raw.gestures);
    if (!gLibrary.load()) {

```

```

        finish();
    }

    GestureOverlayView gOverlay = findViewById(R.id.gOverlay);
    gOverlay.addOnGesturePerformedListener(this);
}

```

28.13 Implementing the onGesturePerformed Method

All that remains before an initial test run of the application can be performed is to implement the *OnGesturePerformed* callback method. This is the method which will be called when a gesture is performed on the *GestureOverlayView* instance:

```

package com.ebookfrenzy.customgestures;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.gesture.GestureLibraries;
import android.gesture.GestureLibrary;
import android.gesture.GestureOverlayView;
import android.gesture.GestureOverlayView.OnGesturePerformedListener;
import android.gesture.Prediction;
import android.widget.Toast;
import android.gesture.Gesture;
import java.util.ArrayList;

public class CustomGesturesActivity extends AppCompatActivity
    implements OnGesturePerformedListener {

    private GestureLibrary gLibrary;
    .
    .

    public void onGesturePerformed(GestureOverlayView overlay,
    Gesture
        gesture) {
        ArrayList<Prediction> predictions =
            gLibrary.recognize(gesture);

        if (predictions.size() > 0 && predictions.get(0).score > 1.0)
        {

            String action = predictions.get(0).name;

```



```

        Toast.makeText(this, action, Toast.LENGTH_SHORT).show();
    }
}
.
.
.
}

```

When a gesture on the gesture overlay view object is detected by the Android runtime, the *onGesturePerformed* method is called. Passed through as arguments are a reference to the *GestureOverlayView* object on which the gesture was detected together with an object of type *Gesture*. The *Gesture* class is designed to hold the information that defines a specific gesture (essentially a sequence of timed points on the screen depicting the path of the strokes that comprise a gesture).

The *Gesture* object is passed through to the *recognize()* method of our *gLibrary* instance, the purpose of which is to compare the current gesture with each gesture loaded from the gestures file. Once this task is complete, the *recognize()* method returns an *ArrayList* object containing a *Prediction* object for each comparison performed. The list is ranked in order from the best match (at position 0 in the array) to the worst. Contained within each prediction object is the name of the corresponding gesture from the gestures file and a prediction score indicating how closely it matches the current gesture.

The code in the above method, therefore, takes the prediction at position 0 (the closest match) makes sure it has a score of greater than 1.0 and then displays a Toast message (an Android class designed to display notification pop ups to the user) displaying the name of the matching gesture.

28.14 Testing the Application

Build and run the application on either an emulator or a physical Android device and perform the circle and swipe gestures on the display. When performed, the toast notification should appear containing the name of the gesture that was performed. Note that when a gesture is recognized, it is outlined on the display with a bright yellow line while gestures about which the overlay is uncertain appear as a faded yellow line. While useful during development, this is probably not ideal for a real world application. Clearly, therefore, there is still some more configuration work to do.

28.15 Configuring the GestureOverlayView

By default, the `GestureOverlayView` is configured to display yellow lines during gestures. The color used to draw recognized and unrecognized gestures can be defined via the `android:gestureColor` and `android:uncertainGestureColor` attributes. For example, to hide the gesture lines, modify the `activity_custom_gestures.xml` file in the example project as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    xmlns:android="http://schemas.android.com/apk/res/android">

    <android.gesture.GestureOverlayView
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:id="@+id/gOverlay"
        android:layout_gravity="center_horizontal"
        android:gestureColor="#00000000"
        android:uncertainGestureColor="#00000000" >
    </android.gesture.GestureOverlayView>
</LinearLayout>
```

On re-running the application, gestures should now be invisible (since they are drawn in white on the white background of the `LinearLayout` view).

28.16 Intercepting Gestures

The `GestureOverlayView` is, as previously described, a transparent overlay that may be positioned over the top of other views. This leads to the question as to whether events intercepted by the gesture overlay should then be passed on to the underlying views when a gesture has been recognized. This is controlled via the `android:eventsInterceptionEnabled` property of the `GestureOverlayView` instance. When set to true, the gesture events are not passed to the underlying views when a gesture is recognized. This can be a particularly useful setting when gestures are being performed over a view that might be configured to scroll in response to certain gestures. Setting this property to *true* will avoid gestures also being interpreted as instructions to the underlying view to scroll in a particular direction.

28.17 Detecting Pinch Gestures

Before moving on from touch handling in general and gesture recognition in particular, the last topic of this chapter is that of handling pinch gestures. While it is possible to create and detect a wide range of gestures using the steps outlined in the previous sections of this chapter it is, in fact, not possible to detect a pinching gesture (where two fingers are used in a stretching and pinching motion, typically to zoom in and out of a view or image) using the techniques discussed so far.

The simplest method for detecting pinch gestures is to use the Android *ScaleGestureDetector* class. In general terms, detecting pinch gestures involves the following three steps:

1. Declaration of a new class which implements the *SimpleOnScaleGestureListener* interface including the required *onScale()*, *onScaleBegin()* and *onScaleEnd()* callback methods.
2. Creation of an instance of the *ScaleGestureDetector* class, passing through an instance of the class created in step 1 as an argument.
3. Implementing the *onTouchEvent()* callback method on the enclosing activity which, in turn, calls the *onTouchEvent()* method of the *ScaleGestureDetector* class.

In the remainder of this chapter, we will create a very simple example designed to demonstrate the implementation of pinch gesture recognition.

28.18A Pinch Gesture Example Project

Create a new project in Android Studio, entering *PinchExample* into the Application name field and *ebookfrenzy.com* as the Company Domain setting before clicking on the *Next* button.

On the form factors screen, enable the *Phone and Tablet* option and set the minimum SDK setting to API 14: Android 4.0 (IceCreamSandwich). Continue to proceed through the screens, requesting the creation of an Empty Activity named *PinchExampleActivity* with a layout resource file named *activity_pinch_example*.

Within the *activity_pinch_example.xml* file, select the default *TextView* object and use the Attributes tool window to set the ID to *myTextView*.

Locate and load the *PinchExampleActivity.java* file into the Android Studio

editor and modify the file as follows:

```
package com.ebookfrenzy.pinchexample;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.MotionEvent;
import android.view.ScaleGestureDetector;
import
android.view.ScaleGestureDetector.SimpleOnScaleGestureListener;
import android.widget.TextView;

public class PinchExampleActivity extends AppCompatActivity {

    TextView scaleText;
    ScaleGestureDetector scaleGestureDetector;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_pinch_example);

        scaleText = (TextView)findViewById(R.id.myTextView);

        scaleGestureDetector =
            new ScaleGestureDetector(this,
                new MyOnScaleGestureListener());
    }

    @Override
    public boolean onTouchEvent(MotionEvent event) {
        scaleGestureDetector.onTouchEvent(event);
        return true;
    }

    public class MyOnScaleGestureListener extends
        SimpleOnScaleGestureListener {

        @Override
        public boolean onScale(ScaleGestureDetector detector) {

            float scaleFactor = detector.getScaleFactor();

            if (scaleFactor > 1) {
```

```

        scaleText.setText("Zooming Out");
    } else {
        scaleText.setText("Zooming In");
    }
    return true;
}

@Override
public boolean onScaleBegin(ScaleGestureDetector detector) {
    return true;
}

@Override
public void onScaleEnd(ScaleGestureDetector detector) {

}
}
.
.
.
}

```

The code declares a new class named `MyOnScaleGestureListener` which extends the `Android SimpleOnScaleGestureListener` class. This interface requires that three methods (*onScale()*, *onScaleBegin()* and *onScaleEnd()*) be implemented. In this instance the *onScale()* method identifies the scale factor and displays a message on the text view indicating the type of pinch gesture detected.

Within the *onCreate()* method, a reference to the text view object is obtained and assigned to the `scaleText` variable. Next, a new *ScaleGestureDetector* instance is created, passing through a reference to the enclosing activity and an instance of our new *MyOnScaleGestureListener* class as arguments. Finally, an *onTouchEvent()* callback method is implemented for the activity, which simply calls the corresponding *onTouchEvent()* method of the *ScaleGestureDetector* object, passing through the *MotionEvent* object as an argument.

Compile and run the application on an emulator or physical Android device and perform pinching gestures on the screen, noting that the text view displays either the zoom in or zoom out message depending on the pinching motion. Pinching gestures may be simulated within the emulator by holding

down the Ctrl (or Cmd) key and clicking and dragging the mouse pointer as shown in [Figure 28-4](#):

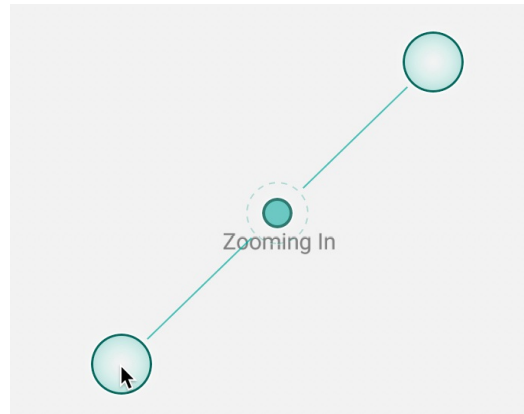


Figure 28-4

28.19 Summary

A gesture is essentially the motion of points of contact on a touch screen involving one or more strokes and can be used as a method of communication between user and application. Android allows gestures to be designed using the Gesture Builder application. Once created, gestures can be saved to a gestures file and loaded into an activity at application runtime using the GestureLibrary.

Gestures can be detected on areas of the display by overlaying existing views with instances of the transparent *GestureOverlayView* class and implementing an *OnGesturePerformedListener* event listener. Using the GestureLibrary, a ranked list of matches between a gesture performed by the user and the gestures stored in a gestures file may be generated, using a prediction score to decide whether a gesture is a close enough match.

Pinch gestures may be detected through the implementation of the ScaleGestureDetector class, an example of which was also provided in this chapter.