24

Handling Advanced User Input

Users interact with Android devices in many ways, including using keyboards, trackballs, touch-screen gestures, and even voice. Different devices support different input methods and have different hardware. For example, certain devices have hardware keyboards, and others rely only upon software keyboards. In this chapter, you learn about the different input methods available to developers and how you can use them to great effect within your applications.

Working with Textual Input Methods

The Android SDK includes input method framework classes that enable interested developers to use powerful input methods as well as create their own input methods, such as custom software keyboards and other Input Method Editors (IMEs). Users can download custom IMEs to use on their devices. For example, there's nothing stopping a developer from creating a custom keyboard with Lord of the Rings-style Elvish characters, smiley faces, or Greek symbols.

Tip

Most device settings related to input methods are available under the Settings, Language & Keyboard menu. Here users can select the language as well as configure the custom user dictionary and make changes to how their keyboards function. The user can change the input method on the device by press-and-holding an `EditText` control, for example. A context menu comes up, allowing the user to change the input method (Android keyboard is usually the default).

Working with Software Keyboards

Because text input methods are locale-based (different countries use different alphabets and keyboards) and situational (numeric vs. alphabetic vs. special keys), the Android platform has trended toward software keyboards as opposed to relying on hardware manufacturers to deliver specialized hardware keyboards.

Choosing the Appropriate Software Keyboard

The Android platform has a number of software keyboards available for use. One of the easiest ways to enable your users to enter data efficiently is to specify the type of input expected in each text input field.

Tip

Many of the code examples provided in this section are taken from the SimpleTextInput-Types application. The source code for this application is provided for download on the book website.

For example, to specify an `EditText` that should take only capitalized textual input, you could set the `inputType` attribute as follows:

```
<EditText android:layout_height="wrap_content"
    android:layout_width="fill_parent"
    android:inputType="text|textCapCharacters">
</EditText>
```

Figure 24.1 shows a number of `EditText` controls with different `inputType` configurations.

Figure 24.1 EditText Controls with different input types.

The input type dictates which software keyboard is used by default and it enforces appropriate rules, such as limiting input to certain characters.

Figure 24.2 (left) illustrates what the software keyboard looks like for an `EditText` control with its `inputType` attribute set to all capitalized text input. Note how the software keyboard keys are all capitalized. If you were to set the `inputType` to `textCapWords` instead, the keyboard switches to lowercase after the first letter of each word and then back to uppercase after a space. Figure 24.2 (middle) illustrates what the software keyboard looks like for an `EditText` control with its `inputType` attribute set to `number`. Figure 24.2 (right) illustrates what the software keyboard looks like for an `EditText` control with its `inputType` attribute set to textual input, where each sentence begins with a capital letter and the text can be multiple lines.

Figure 24.2 The software keyboards associated with specific input types.

Depending on the user's keyboard settings (specifically, if the user has enabled the Show Suggestions and Auto-complete options in the Android Keyboard settings of his device), the user might also see suggested words or spelling fixes while typing.

For a complete list of `inputType` attribute values and their uses, see http://developer.android.com/reference/android/R.attr.html#inputType.

Tip

You can also have your `Activity` react to the display of software keyboards (to adjust where fields are displayed, for example) by requesting the `WindowManager` as a system service and modifying the layout parameters associated with the `softInputMode` field.

For more fine-tuned control over input methods, see the `android.view.inputmethod.InputMethodManager` class.

Providing Custom Software Keyboards

If you are interested in developing your own software keyboards, we highly recommend the following references:

- IMEs are implemented as an Android service. Begin by reviewing the Android packages called `android.inputmethodservice` and `android.view.inputmethod`, which can be used to implement custom input methods.
- The SoftKeyboard sample application in the Android SDK provides an implementation of a software keyboard.
- The Android Developer technical articles on onscreen input methods (http://developer.android.com/resources/articles/on-screen-inputs.html) and creating an input method (http://developer.android.com/resources/articles/creating-input-method.html).

Working with Text Prediction and User Dictionaries

Text prediction is a powerful and flexible feature available on Android devices. We've already talked about many of these technologies in other parts of this book, but they merit mentioning in this context as well.

- In Chapter 7, "Exploring User Interface Screen Elements," you learned how to use `AutoCompleteTextView` and `MultiAutoCompleteTextView` controls to help users input common words and strings.
- In Chapter 10, "Using Android Data and Storage APIs," you learned how to tie an `AutoCompleteTextView` control to an underlying SQLite database table.
- In Chapter 11, "Sharing Data Between Applications with Content Providers," you learned about the `UserDictionary` content provider (`android.provider.UserDictionary`), which can be used to add words for the user's custom dictionary of commonly used words.

Exploring the Accessibility Framework

The Android SDK includes numerous features and services for the benefit of users with visual and hearing impairments. Those users without such impairments also benefit from these features, especially when they are not paying complete attention to the device (such as when driving). Many of the most powerful accessibility features were added in Android 1.6 and 2.0, so check the API level for a specific class or method before using it within your application. Some of the accessibility features available within the Android SDK include

- The Speech Recognition Framework.
- The Text-To-Speech (TTS) Framework.

- The ability to enable haptic feedback (that vibration you feel when you press a button, rather like a rumble pack game controller) on any `View` object (API Level 3 and higher). See the `setHapticFeedbackEnabled()` method of the `View` class.
- The ability to set associated metadata, such as a text description of an `ImageView` control on any `View` object (API Level 4 and higher). This feature is often very helpful for the visually impaired. See the `setContentDescription()` method of the `View` class.
- The ability to create and extend accessibility applications in conjunction with the Android Accessibility framework. See the following packages to get started writing accessibility applications: `android.accessibilityservice` and `android.view.accessibility`. There are also a number of accessibility applications, such as KickBack, SoundBack, and TalkBack, which ship with the platform. For more information, see the device settings under Settings, Accessibility.

Tip

Give some thought to providing accessibility features, such as providing View metadata, within your applications. There's really no excuse for not doing so. Your users appreciate these small details, which make all the difference in terms of whether or not certain users can use your application at all. Also, make sure your quality assurance team verifies accessibility features as part of their testing process.

Because speech recognition and Text-To-Speech applications are all the rage, and their technologies are often used for navigation applications (especially because many states are passing laws making driving while using a mobile device without hands-free operation illegal), let's look at these two technologies in a little more detail.

Android applications can leverage speech input and output. Speech input can be achieved using speech recognition services and speech output can be achieved using Text-To-Speech services. Not all devices support these services. However, certain types of applications—most notably hands-free applications such as directional navigation—often benefit from the use of these types of input.

Speech services are available within the Android SDK in the `android.speech` package. The underlying services that make these technologies work might vary from device to device; some services might require a network connection to function properly.

Tip

Many of the code examples provided in this section are taken from the SimpleSpeech application. The source code for this application is provided for download on the book website. Speech services are best tested on a real Android device. We used an HTC Nexus One running Android 2.2 in our testing.

Leveraging Speech Recognition Services

You can enhance an application with speech recognition support by using the speech recognition framework provided within the Android SDK. Speech recognition involves speaking into the device microphone and enabling the software to detect and interpret

that speech and translate it into a string. Speech recognition services are intended for use with short command-like phrases without pauses, not for long dictation. If you want more robust speech recognition, you need to implement your own solution.

On Android SDK 2.1 and higher, access to speech recognition is built in to most pop-up keyboards. Therefore, an application might already support speech recognition, to some extent, without any changes. However, directly accessing the recognizer can allow for more interesting spoken-word control over applications.

You can use the `android.speech.RecognizerIntent` intent to launch the built-in speech recorder. This launches the recorder (shown in Figure 24.3), allowing the user to record speech.

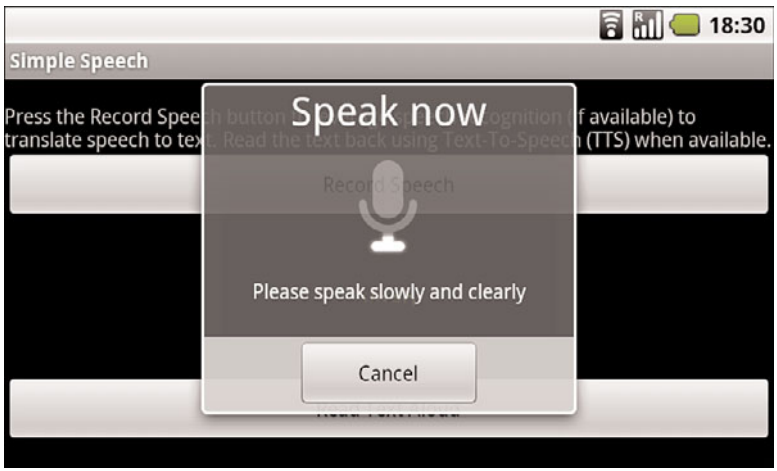

Figure 24.3 Recording speech with the **`RecognizerIntent`**.

The sound file is sent to an underlying recognition server for processing, so this feature is not really practical for devices that don't have a reasonable network connection. You can then retrieve the results of the speech recognition processing and use them within your application. Note that you might receive multiple results for a given speech segment.

Note

Speech recognition technology is continually evolving and improving. Be sure to enunciate clearly when speaking to your device. Sometimes it might take several tries before the speech recognition engine interprets your speech correctly.

The following code demonstrates how an application could be enabled to record speech using the `RecognizerIntent` intent:

```
public class SimpleSpeechActivity extends Activity
{
    private static final int VOICE_RECOGNITION_REQUEST = 1;
```

```
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }
    public void recordSpeech(View view) {
        Intent intent =
            new Intent(RecognizerIntent.ACTION_RECOGNIZE_SPEECH);
        intent.putExtra(RecognizerIntent.EXTRA_LANGUAGE_MODEL,
            RecognizerIntent.LANGUAGE_MODEL_FREE_FORM);
        intent.putExtra(RecognizerIntent.EXTRA_PROMPT,
            "Please speak slowly and clearly");
        startActivityForResult(intent, VOICE_RECOGNITION_REQUEST);
    }

    @Override
    protected void onActivityResult(int requestCode,
        int resultCode, Intent data) {
        if (requestCode == VOICE_RECOGNITION_REQUEST &&
            resultCode == RESULT_OK) {
            ArrayList<String> matches = data.getStringArrayListExtra(
                RecognizerIntent.EXTRA_RESULTS);
            TextView textSaid = (TextView) findViewById(R.id.TextSaid);
            textSaid.setText(matches.get(0));
        }
        super.onActivityResult(requestCode, resultCode, data);
    }
}
```

In this case, the intent is initiated through the click of a `Button` control, which causes the `recordSpeech()` method to be called. The `RecognizerIntent` is configured as follows:

- The intent action is set to `ACTION_RECOGNIZE_SPEECH` in order to prompt the user to speak and send that sound file in for speech recognition.
- An intent extra called `EXTRA_LANGUAGE_MODEL` is set to `LANGUAGE_MODEL_FREE_FORM` to simply perform standard speech recognition. There is also another language model especially for web searches called `LANGUAGE_MODEL_WEB_SEARCH`.
- An intent extra called `EXTRA_PROMPT` is set to a string to display to the user during speech input.

After the `RecognizerIntent` object is configured, the intent can be started using the `startActivityForResult()` method, and then the result is captured in the `onActivityResult()` method. The resulting text is then displayed in the `TextView` control called `TextSaid`. In this case, only the first result provided in the results is displayed to the user. So, for example, the user could press the button initiating the `recordSpeech()`

method, say "We're going to need a bigger boat," and that text is then displayed in the application's `TextView` control, as shown in Figure 24.4.

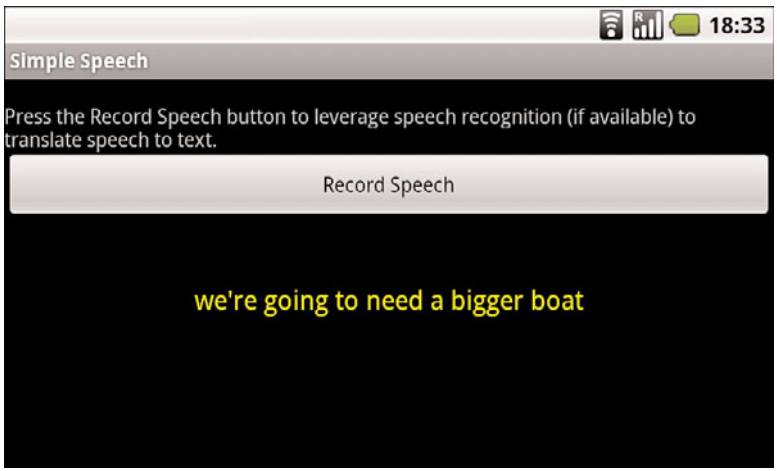

Figure 24.4 The text string resulting from the `RecognizerIntent`.

Leveraging Text-To-Speech Services

The Android platform includes a TTS engine (`android.speech.tts`) that enables devices to perform speech synthesis. You can use the TTS engine to have your applications "read" text to the user. You might have seen this feature used frequently with location-based services (LBS) applications that allow for hands-free directions. Other applications use this feature for users who have reading or sight problems. The synthesized speech can be played immediately or saved to an audio file, which can be treated like any other audio file.

> **Note**
>
> To provide TTS services to users, an Android device must have both the TTS engine (available in Android SDK 1.6 and higher) and the appropriate language resource files. In some cases, the user must install the appropriate language resource files (assuming that the user has space for them) from a remote location. The users can install the language resource files by going to Settings, Voice Input & Output Settings, Text-to-Speech, Install Voice Data. Unlike some other settings pages, this one doesn't have a specific intent action defined under `android.provider.Settings`. You might also need to do this on your devices. Additionally, the application can verify that the data is installed correctly or trigger the installation if it's not.

For a simple example, let's have the device read back the text recognized in our earlier speech recognition example. First, we must modify the activity to implement the `TextToSpeech.OnInitListener` interface, as follows:

```
public class SimpleSpeechActivity extends Activity
    implements TextToSpeech.OnInitListener
{
    // class implementation
}
```

Next, you need to initialize TTS services within your activity:

```
TextToSpeech mTts = new TextToSpeech(this, this);
```

Initializing the TTS engine happens asynchronously. The `TextToSpeech.OnInitListener` interface has only one method, `onInit()`, that is called when the TTS engine has finished initializing successfully or unsuccessfully. Here is an implementation of the `onInit()` method:

```
@Override
public void onInit(int status) {
    Button readButton = (Button) findViewById(R.id.ButtonRead);
    if (status == TextToSpeech.SUCCESS) {
        int result = mTts.setLanguage(Locale.US);
        if (result == TextToSpeech.LANG_MISSING_DATA
            || result == TextToSpeech.LANG_NOT_SUPPORTED) {
            Log.e(DEBUG_TAG, "TTS Language not available.");
            readButton.setEnabled(false);
        } else {
            readButton.setEnabled(true);
        }
    } else {
        Log.e(DEBUG_TAG, "Could not initialize TTS Engine.");
        readButton.setEnabled(false);
    }
}
```

We use the `onInit()` method to check the status of the TTS engine. If it was initialized successfully, the `Button` control called `readButton` is enabled; otherwise, it is disabled. The `onInit()` method is also the appropriate time to configure the TTS engine. For example, you should set the language used by the engine using the `setLanguage()` method. In this case, the language is set to American English. The voice used by the TTS engine uses American pronunciation.

Note

The Android TTS engine supports a variety of languages, including English (in American or British accents), French, German, Italian, and Spanish. You could just as easily have enabled British English pronunciation using the following language setting in the `onInit()` method implementation instead:

```
int result = mTts.setLanguage(Locale.UK);
```

> We amused ourselves trying to come up with phrases that illustrate how the American and British English TTS services differ. The best phrase we came up with was: *"We adjusted our schedule to search for a vase of herbs in our garage."*
>
> Feel free to send us your favorite locale-based phrases, and we will post them on the book website. Also, any amusing misinterpretations of the voice recognition are also welcome (for example, we often had "our garage" come out as "nerd haha").

Finally, you are ready to actually convert some text into a sound file. In this case, we grab the text string currently stored in the `TextView` control (where we set using speech recognition in the previous section) and pass it to TTS using the `speak()` method:

```
public void readText(View view) {
    TextView textSaid = (TextView) findViewById(R.id.TextSaid);
    mTts.speak((String) textSaid.getText(),
        TextToSpeech.QUEUE_FLUSH, null);
}
```

The `speak()` method takes three parameters: the string of text to say, the queuing strategy and the speech parameters. The queuing strategy can either be to add some text to speak to the queue or to flush the queue—in this case, we use the `QUEUE_FLUSH` strategy, so it is the only speech spoken. No special speech parameters are set, so we simply pass in `null` for the third parameter. Finally, when you are done with the `TextToSpeech` engine (such as in your activity's `onDestroy()` method), make sure to release its resources using the `shutdown()` method:

```
mTts.shutdown();
```

Now, if you wire up a `Button` control to call the `readText()` method when clicked, you have a complete implementation of TTS. When combined with the speech recognition example discussed earlier, you can develop an application that can record a user's speech, translate it into a string, display that string on the screen, and then read that string back to the user. In fact, that is exactly what the sample project called SimpleSpeech does.

Working with Gestures

Android devices often rely upon touch screens for user input. Users are now quite comfortable using common finger gestures to operate their devices. Android applications can detect and react to one-finger (single-touch) and two-finger (multi-touch) gestures.

> **Note**
>
> Even early Android devices supported simple single touch gestures. Support for multi-touch gestures was added in the Android 2.2 SDK and is available only on devices with capacitive touch screen hardware.

One of the reasons that gestures can be a bit tricky is that a gesture can be made of multiple touch events, or motions. Different sequences of motion add up to different gestures. For example, a fling gesture involves the user pressing his finger down on the screen,

swiping across the screen, and lifting his finger up off the screen while the swipe is still in motion (that is, without slowing down to stop before lifting his finger). Each of these steps can trigger motion events that applications can react to.

Detecting User Motions Within a View

By now you've come to understand that Android application user interfaces are built using different types of `View` controls. Developers can handle gestures much like they do click events within a `View` control using the `setOnClickListener()` and `setOnLongClickListener()` methods. Instead, the `onTouchEvent()` callback method is used to detect that some motion has occurred within the `View` region.

The `onTouchEvent()` callback method has a single parameter: a `MotionEvent` object. The `MotionEvent` object contains all sorts of details about what kind of motion is occurring within the `View`, enabling the developer to determine what sort of gesture is happening by collecting and analyzing many consecutive `MotionEvent` objects. You could use all of the `MotionEvent` data to recognize and detect every kind of gesture you could possibly imagine. Alternately, you can use built-in gesture detectors provided in the Android SDK to detect common user motions in a consistent fashion. Android currently has two different classes that can detect navigational gestures:

- The `GestureDetector` class can be used to detect common single-touch gestures.
- The `ScaleGestureDetector` can be used to detect multi-touch scale gestures.

It is likely that more gesture detectors will be added in future versions of the Android SDK. You can also implement your own gesture detectors to detect any gestures not supported by the built-in gesture detectors. For example, you might want to create a two-fingered rotate gesture to, say, rotate an image or a three-fingered swipe gesture that brings up an option menu.

In addition to common navigational gestures, you can use the `android.gesture` package with the `GestureOverlayView` to recognize command-like gestures. For instance, you could create an S-shaped gesture that brings up a search, or a zig-zag gesture that clears a screen on a drawing app. Tools are available for recording and creating libraries of this style gesture. As it uses an overlay for detection, it isn't well suited for all types of applications. This package was introduced in API Level 4.

> **Warning**
>
> The type and sensitivity of the touch screen can vary by device. Different devices can detect different numbers of touch points simultaneously, which affects the complexity of gestures you can support.

Handling Common Single-Touch Gestures

Introduced in API Level 1, the `GestureDetector` class can be used to detect gestures made by a single finger. Some common single finger gestures supported by the `GestureDetector` class include:

- `onDown`: Called when the user first presses on the touch screen.
- `onShowPress`: Called after the user first presses the touch screen but before he lifts his finger or moves it around on the screen; used to visually or audibly indicate that the press has been detected.
- `onSingleTapUp`: Called when the user lifts up (using the up `MotionEvent`) from the touch screen as part of a single-tap event.
- `onSingleTapConfirmed`: Called when a single-tap event occurs.
- `onDoubleTap`: Called when a double-tap event occurs.
- `onDoubleTapEvent`: Called when an event within a double-tap gesture occurs, including any down, move, or up `MotionEvent`.
- `onLongPress`: Similar to `onSingleTapUp`, but called if the user holds down his finger long enough to not be a standard click but also without any movement.
- `onScroll`: Called after the user presses and then moves his finger in a steady motion before lifting his finger. This is commonly called dragging.
- `onFling`: Called after the user presses and then moves his finger in an accelerating motion before lifting it. This is commonly called a flick gesture and usually results in some motion continuing after the user lifts his finger.

You can use the interfaces available with the `GestureDetector` class to listen for specific gestures such as single and double taps (see `GestureDetector.OnDoubleTapListener`), as well as scrolls and flings (see `GestureDetector.OnGestureListener`). The scrolling gesture involves touching the screen and moving your finger around on it. The fling gesture, on the other hand, causes (though not automatically) the object to continue to move even after the finger has been lifted from the screen. This gives the user the impression of throwing or flicking the object around on the screen.

Tip

You can use the `GestureDetector.SimpleOnGestureListener` class to listen to any and all of the gestures recognized by the `GestureDetector`.

Let's look at a simple example. Let's assume you have a game screen that enables the user to perform gestures to interact with a graphic on the screen. We can create a custom `View` class called `GameAreaView` that can dictate how a bitmap graphic moves around within the game area based upon each gesture. The `GameAreaView` class can use the `onTouchEvent()` method to pass along `MotionEvent` objects to a `GestureDetector`. In this way, the `GameAreaView` can react to simple gestures, interpret them, and make the appropriate changes to the bitmap, including moving it from one location to another on the screen.

Tip

How the gestures are interpreted and what actions they cause is completely up to the developer. You could, for example, interpret a fling gesture and make the bitmap graphic disappear... but does that make sense? Not really. It's important to always make the gesture jive well with the resulting operation within the application so that users are not confused. Users are now accustomed to specific screen behavior based on certain gestures, so it's best to use the expected convention, too.

In this case, the `GameAreaView` class interprets gestures as follows:

- A double-tap gesture causes the bitmap graphic to return to its initial position.
- A scroll gesture causes the bitmap graphic to "follow" the motion of the finger.
- A fling gesture causes the bitmap graphic to "fly" in the direction of the fling.

Tip

Many of the code examples provided in this section are taken from the SimpleGestures application. The source code for this application is provided for download on the book website.

To make these gestures work, the `GameAreaView` class needs to include the appropriate gesture detector, which triggers any operations upon the bitmap graphic. Based upon the specific gestures detected, the `GameAreaView` class must perform all translation animations and other graphical operations applied to the bitmap. To wire up the `GameAreaView` class for gesture support, we need to implement several important methods:

- The class constructor must initialize any gesture detectors and bitmap graphics.
- The `onTouchEvent()` method must be overridden to pass the `MotionEvent` data to the gesture detector for processing.
- The `onDraw()` method must be overridden to draw the bitmap graphic in the appropriate position at any time.
- Various methods are needed to perform the graphics operations required to make a bitmap move around on the screen, fly across the screen, reset its location based upon the data provided by the specific gesture.

All these tasks are handled by our `GameAreaView` class definition:

```
public class GameAreaView extends View {

    private static final String DEBUG_TAG =
        "SimpleGesture->GameAreaView";
    private GestureDetector gestures;
    private Matrix translate;
    private Bitmap droid;
    private Matrix animateStart;
    private Interpolator animateInterpolator;
    private long startTime;
    private long endTime;
```

```
private float totalAnimDx;
private float totalAnimDy;

public GameAreaView(Context context, int iGraphicResourceId) {
    super(context);
    translate = new Matrix();
    GestureListener listener = new GestureListener(this);
    gestures = new GestureDetector(context, listener, null, true);
    droid = BitmapFactory.decodeResource(getResources(),
        iGraphicResourceId);
}

@Override
public boolean onTouchEvent(MotionEvent event) {
    boolean retVal = false;
    retVal = gestures.onTouchEvent(event);
    return retVal;
}

@Override
protected void onDraw(Canvas canvas) {
    Log.v(DEBUG_TAG, “onDraw");
    canvas.drawBitmap(droid, translate, null);
}

public void onResetLocation() {
    translate.reset();
    invalidate();
}

public void onMove(float dx, float dy) {
    translate.postTranslate(dx, dy);
    invalidate();
}

public void onAnimateMove(float dx, float dy, long duration) {
    animateStart = new Matrix(translate);
    animateInterpolator = new OvershootInterpolator();
    startTime = System.currentTimeMillis();
    endTime = startTime + duration;
    totalAnimDx = dx;
    totalAnimDy = dy;
    post(new Runnable() {
        @Override
        public void run() {
            onAnimateStep();
        }
```

```
        });
    }

    private void onAnimateStep() {
        long curTime = System.currentTimeMillis();
        float percentTime = (float) (curTime - startTime) /
            (float) (endTime - startTime);
        float percentDistance = animateInterpolator
            .getInterpolation(percentTime);
        float curDx = percentDistance * totalAnimDx;
        float curDy = percentDistance * totalAnimDy;
        translate.set(animateStart);
        onMove(curDx, curDy);

        if (percentTime < 1.0f) {
            post(new Runnable() {
                @Override
                public void run() {
                    onAnimateStep();
                }
            });
        }
    }
}
```

As you can see, the `GameAreaView` class keeps track of where the bitmap graphic should be drawn at any time. The `onTouchEvent()` method is used to capture motion events and pass them along to a gesture detector whose `GestureListener` we must implement as well (more on this in a moment). Typically, each method of the `GameAreaView` applies some operation to the bitmap graphic and then calls the `invalidate()` method, forcing the view to be redrawn. Now we turn our attention to the methods required to implement specific gestures:

- For double-tap gestures, we implement a method called `onResetLocation()` to draw the bitmap graphic in its original location.
- For scroll gestures, we implement a method called `onMove()` to draw the bitmap graphic in a new location. Note that scrolling can occur in any direction—it simply refers to a finger swipe on the screen.
- For fling gestures, things get a little tricky. To animate motion on the screen smoothly, we used a chain of asynchronous calls and a built-in Android interpolator to calculate the location to draw the graphic based upon how long it had been since the animation started. See the `onAnimateMove()` and `onAnimateStep()` methods for the full implementation of fling animation.

Now we need to implement our `GestureListener` class to interpret the appropriate gestures and call the `GameAreaView` methods we just implemented. Here's an implementation of the `GestureListener` class that our `GameAreaView` class can use:

```
private class GestureListener extends
    GestureDetector.SimpleOnGestureListener  {

    GameAreaView view;

    public GestureListener(GameAreaView view) {
        this.view = view;
    }

    @Override
    public boolean onDown(MotionEvent e) {
        return true;
    }

    @Override
    public boolean onFling(MotionEvent e1, MotionEvent e2,
        final float velocityX, final float velocityY) {
        final float distanceTimeFactor = 0.4f;
        final float totalDx = (distanceTimeFactor * velocityX / 2);
        final float totalDy = (distanceTimeFactor * velocityY / 2);

        view.onAnimateMove(totalDx, totalDy,
            (long) (1000 * distanceTimeFactor));
        return true;
    }

    @Override
    public boolean onDoubleTap(MotionEvent e) {
        view.onResetLocation();
        return true;
    }

    @Override
    public boolean onScroll(MotionEvent e1, MotionEvent e2,
        float distanceX, float distanceY) {
        view.onMove(-distanceX, -distanceY);
        return true;
    }
}
```

Note that you must return true for any gesture or motion event that you want to detect. Therefore, you must return true in the `onDown()` method as it happens at the beginning

of a scroll-type gesture. Most of the implementation of the `GestureListener` class methods involves our interpretation of the data for each gesture. For example:

- We react to double taps by resetting the bitmap to its original location using the `onResetLocation()` method of our `GameAreaView` class.
- We use the distance data provided in the `onScroll()` method to determine the direction to use in the movement to pass into the `onMove()` method of the `GameAreaView` class.
- We use the velocity data provided in the `onFling()` method to determine the direction and speed to use in the movement animation of the bitmap. The `timeDistanceFactor` variable with a value of 0.4 is subjective, but gives the resulting slide-to-a-stop animation enough time to be visible but is short enough to be controllable and responsive. You could think of it as a high-friction surface. This information is used by the animation sequence implemented within the `onAnimateMove()` method of the `GameAreaView` class.

Now that we have implemented the `GameAreaView` class in its entirety, you can display it on a screen. For example, you might create an `Activity` that has a user interface with a `FrameLayout` control and add an instance of a `GameAreaView` using the `addView()` method. The resulting scroll and fling gestures look something like Figure 24.5.

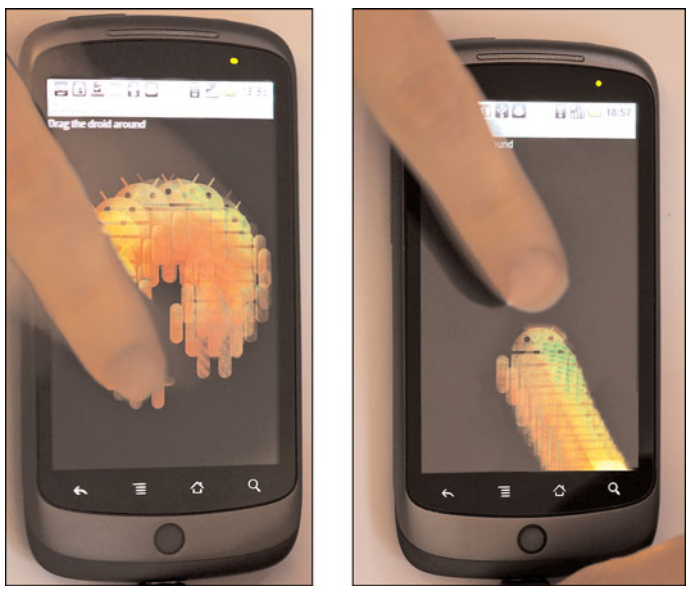

Figure 24.5 Scroll (left) and Fling (right) gestures.

Tip

To support the broadest range of devices, we recommend supporting simple, one-fingered gestures and providing alternate navigational items for devices that don't support multi-touch gestures. However, users are beginning to expect multi-touch gesture support now, so use them where you can and where they make sense. Resistive touch-screens remain somewhat uncommon on lower-end devices.

Handling Common Multi-Touch Gestures

Introduced in API Level 8 (Android 2.2), the `ScaleGestureDetector` class can be used to detect two-fingered scale gestures. The scale gesture enables the user to move two fingers toward and away from each other. When the fingers are moving apart, this is considered scaling up; when the fingers are moving together, this is considered scaling down. This is the "pinch-to-zoom" style often employed by map and photo applications.

Tip

You can use the `ScaleGestureDetector.SimpleOnScaleGestureListener` class to detect scale gestures detected by the `ScaleGestureDetector`.

Let's look at another example. Again, we use the custom view class called `GameAreaView`, but this time we handle the multi-touch scale event. In this way, the `GameAreaView` can react to scale gestures, interpret them, and make the appropriate changes to the bitmap, including growing or shrinking it on the screen.

Tip

Many of the code examples provided in this section are taken from the SimpleMultiTouchGesture application. The source code for this application is provided for download on the book website.

In order to handle scale gestures, the `GameAreaView` class needs to include the appropriate gesture detector: a `ScaleGestureDetector`. The `GameAreaView` class needs to be wired up for scale gesture support in a similar fashion as when we implemented single touch gestures earlier, including initializing the gesture detector in the class constructor, overriding the `onTouchEvent()` method to pass the `MotionEvent` objects to the gesture detector, and overriding the `onDraw()` method to draw the view appropriately as necessary. We also need to update the `GameAreaView` class to keep track of the bitmap graphic size (using a `Matrix`) and provide a helper method for growing or shrinking the graphic. Here is the new implementation of the `GameAreaView` class with scale gesture support:

```
public class GameAreaView extends View {
    private ScaleGestureDetector multiGestures;
    private Matrix scale;
    private Bitmap droid;

    public GameAreaView(Context context, int iGraphicResourceId) {
        super(context);
        scale = new Matrix();
```

```
        GestureListener listener = new GestureListener(this);
        multiGestures = new ScaleGestureDetector(context, listener);
        droid = BitmapFactory.decodeResource(getResources(),
            iGraphicResourceId);
    }

    public void onScale(float factor) {
        scale.preScale(factor, factor);
        invalidate();
    }

    @Override
    protected void onDraw(Canvas canvas) {
        Matrix transform = new Matrix(scale);
        float width = droid.getWidth() / 2;
        float height = droid.getHeight() / 2;
        transform.postTranslate(-width, -height);
        transform.postConcat(scale);
        transform.postTranslate(width, height);
        canvas.drawBitmap(droid, transform, null);
    }

    @Override
    public boolean onTouchEvent(MotionEvent event) {
        boolean retVal = false;
        retVal = multiGestures.onTouchEvent(event);
        return retVal;
    }
}
```

As you can see, the `GameAreaView` class keeps track of what size the bitmap should be at any time using the `Matrix` variable called `scale`. The `onTouchEvent()` method is used to capture motion events and pass them along to a `ScaleGestureDetector` gesture detector. As before, the `onScale()` helper method of the `GameAreaView` applies some scaling to the bitmap graphic and then calls the `invalidate()` method, forcing the view to be redrawn.

Now let's take a look at the `GestureListener` class implementation necessary to interpret the scale gestures and call the `GameAreaView` methods we just implemented. Here's the implementation of the `GestureListener` class:

```
private class GestureListener implements
    ScaleGestureDetector.OnScaleGestureListener {

    GameAreaView view;

    public GestureListener(GameAreaView view) {
        this.view = view;
    }
```

```
    @Override
    public boolean onScale(ScaleGestureDetector detector) {
        float scale = detector.getScaleFactor();
        view.onScale(scale);
        return true;
    }

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

    @Override
    public void onScaleEnd(ScaleGestureDetector detector) {
    }
}
```

Remember that you must return `true` for any gesture or motion event that you Want to detect. Therefore, you must return true in the `onScaleBegin()` method as it happens at the beginning of a scale-type gesture. Most of the implementation of the `GestureListener` methods involves our interpretation of the data for the scale gesture. Specifically, we use the scale factor (provided by the `getScaleFactor()` method) to calculate whether we should shrink or grow the bitmap graphic, and by how much. We pass this information to the `onScale()` helper method we just implemented in the `GameAreaView` class.

Now, if you were to use the `GameAreaView` class within your application, scale gestures might look something like Figure 24.6.

Note

The Android emulator does not currently support multi-touch input. You will have to run and test multi-touch support such as the scale gesture using a device running Android 2.2 or higher.

Making Gestures Look Natural

Gestures can enhance your Android application user interfaces in new, interesting, and intuitive ways. Closely mapping the operations being performed on the screen to the user's finger motion makes a gesture feel natural and intuitive. Making application operations look natural requires some experimentation on the part of the developer. Keep in mind that devices vary in processing power, and this might be a factor in making things seem natural.

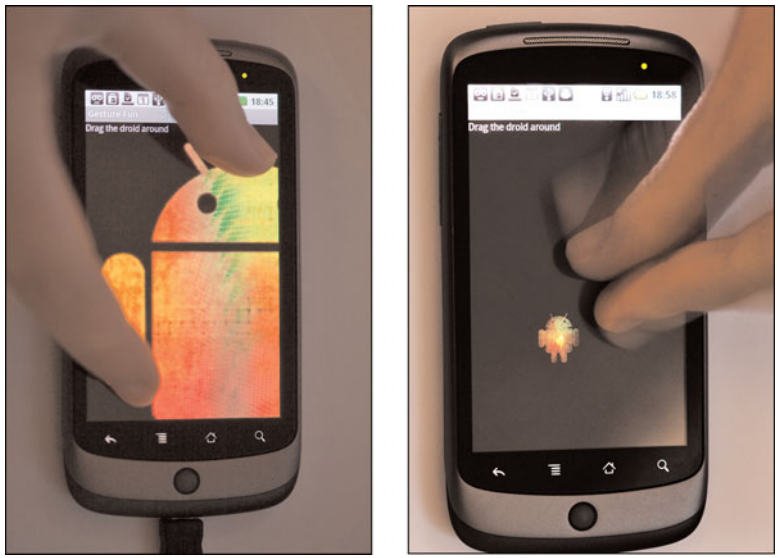

Figure 24.6 Scale up (left) and scale down (right) gestures.

Working with the Trackball

Some Android devices have hardware trackballs, but not all. Developers can handle trackball events within a `View` control in a similar fashion to click events or gestures. To handle trackball events, you can leverage the `View` class method called `onTrackballEvent()`. This method, like a gesture, has a single parameter: a `MotionEvent` object. You can use the `getX()` and `getY()` methods of the `MotionEvent` class to determine the relative movement of the trackball. Optical track-pads such as those available on the Droid Incredible can be supported in the same way.

Tip

If your application requires the device to have a trackball, you should set the `<uses-configuration>` tag to specify that a trackball is required within your application's Android manifest file.

Handling Screen Orientation Changes

Many Android devices on the market today have landscape and portrait modes and can seamlessly transition between these orientations. The Android operating system automatically handles these changes for your application, if you so choose. You can also provide alternative resources, such as different layouts, for portrait and landscape modes (more on

this in Chapter 25, "Targeting Different Device Configurations and Languages"). Also, you can directly access device sensors such as the accelerometer, as we talked about in Chapter 19, "Using Android's Optional Hardware APIs," to capture device orientation along three axes.

However, if you want to listen for simple screen orientation changes programmatically and have your application react to them, you can use the `OrientationEventListener` class to do this within your activity.

Tip

Many of the code examples provided in this section are taken from the SimpleOrientation application. The source code for this application is provided for download on the book website. Orientation changes are best tested on devices, not the emulator.

Implementing orientation event handling within your activity is simple. Simply instantiate an `OrientationEventListener` and provide its implementation. For example, the following activity class called `SimpleOrientationActivity` logs orientation information to LogCat:

```
public class SimpleOrientationActivity extends Activity {

    OrientationEventListener mOrientationListener;

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

        mOrientationListener = new OrientationEventListener(this,
            SensorManager.SENSOR_DELAY_NORMAL) {

            @Override
            public void onOrientationChanged(int orientation) {
                Log.v(DEBUG_TAG,
                    "Orientation changed to " + orientation);
            }
        };

        if (mOrientationListener.canDetectOrientation() == true) {
            Log.v(DEBUG_TAG, "Can detect orientation");
            mOrientationListener.enable();
        } else {
            Log.v(DEBUG_TAG, "Cannot detect orientation");
            mOrientationListener.disable();
        }
    }
```

```
    @Override
    protected void onDestroy() {
        super.onDestroy();
        mOrientationListener.disable();
    }
}
```

You can set the rate to check for orientation changes to a variety of different values. There are other rate values appropriate for game use and other purposes. The default rate, `SENSOR_DELAY_NORMAL`, is most appropriate for simple orientation changes. Other values, such as `SENSOR_DELAY_UI` and `SENSOR_DELAY_GAME`, might make sense for your application.

After you have a valid `OrientationEventListener` object, you can check if it can detect orientation changes using the `canDetectOrientation()` method, and enable and disable the listener using its `enable()` and `disable()` methods.

The `OrientationEventListener` has a single callback method, which enables you to listen for orientation transitions: the `onOrientationChanged()` method. This method has a single parameter, an `integer`. This integer normally represents the device tilt as a number between 0 and 359:

- A result of `ORIENTATION_UNKNOWN` (-1) means the device is flat (perhaps on a table) and the orientation is unknown.
- A result of 0 means the device is in its "normal" orientation, with the top of the device facing in the up direction. (What "normal" means is defined by the manufacturer. You need to test on the device to find out for sure what it means.)
- A result of 90 means the device is tilted at 90 degrees, with the left side of the device facing in the up direction.
- A result of 180 means the device is tilted at 180 degrees, with the bottom side of the device facing in the up direction (upside down).
- A result of 270 means the device is tilted at 270 degrees, with the right side of the device facing in the up direction.

Figure 24.7 shows an example of how the device orientation might read when the device is tilted to the right by 90 degrees.

Warning

Early versions of the Android SDK included a class called `OrientationListener`, which many early developers of the platform used to handle screen orientation transitions. This class is now deprecated, and you should not use it.

Figure 24.7 Orientation of the device as reported by an `OrientationEventListener`.

Summary

The Android platform enables great flexibility when it comes to ways that users can provide input to the device. Developers benefit from the fact that many powerful input methods are built into the view controls themselves, just waiting to be leveraged. Applications can take advantage of built-in input methods, such as software keyboards, or can customize them for special purposes. The Android framework also includes powerful features, such as gesture support, as well as extensive accessibility features, including speech recognition and text-to-speech support. It is important to support a variety of input methods within your applications, as users often have distinct preferences and not all methods are available on all devices.

References and More Information

Android Reference: Faster Orientation Changes:
http://j.mp/9P3yTy
Android Reference: Screen Orientation and Direction:
http://j.mp/b2zY1t