4.2 ADVANCED UI DESIGN

Android provides a flexible framework for UI design that allows apps to display different layouts for different devices, create custom UI widgets, and control aspects of the system UI beyond the apps' window.

4.2.1 Multiple Screens

The goal of this part is to build a UI, which is flexible enough to fit perfectly on any screen and to create different interaction patterns that are optimized for different screen sizes.

To ensure that your layout is flexible and adapts to different screen sizes, you should use "wrap_content" and "match_parent" for the width and height of some view components. If you use "wrap_content", the width or height of the view is set to the minimum size necessary to fit the content within that view, while "match_parent" (also known as "fill_parent" before API level 8) makes the component expand to match the size of its parent view.

```
<item android:state_focused="false">
    <shape>
        <solid android:color="#8f0000f0"/>
        <stroke
            android:width="2dp"
            android:color="#fad3cf" />
        <corners
            android:topRightRadius="5dp"
            android:bottomLeftRadius="5dp"
            android:topLeftRadius="0dp"
            android:bottomRightRadius="0dp"
            />
        <padding
            android:left="10dp"
            android:top="10dp"
            android:right="10dp"
            android:bottom="10dp" />
    </shape>
</item>
```

Figure 4.15 Pressed state of the button.

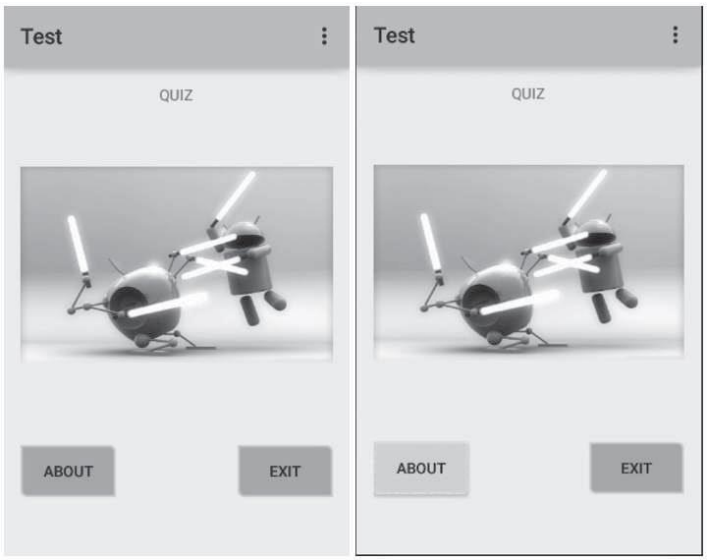

Figure 4.16 Running result of the button selector.

You can construct fairly complex layouts using nested instances of LinearLayout and combinations of "wrap_content" and "match_parent" sizes. However, LinearLayout does not allow you to precisely control the spacial relationships of child views; views in a Lin- earLayout simply line up side by side. If you need child views to be oriented in variations other than a straight line, a better solution is often to use a RelativeLayout, which allows you to specify your layout in terms of the special relationships between components. For instance, you can align one child view on the left side and another view on the right side of the screen.

Supporting different screen sizes usually means that your image re- sources must also be capable of adapting to different sizes. For example, a button background must fit whichever button shape it is applied to. If you use simple images on components that can change size, you will quickly notice that the results are somewhat less than impressive, since the runtime will stretch or shrink your images uniformly. The solution is using

nine-patch bitmaps, which are specially formatted

PNG files that indicate which areas can and cannot be stretched.

Therefore, when designing bitmaps that will be used on components with variable sizes, always use nine-patches. To convert a bitmap into a nine-patch, you can start with a regular image. Then run it through the draw9patch utility of the Software Development Kit (SDK) (which is located in the tools/ directory), in which you can mark the areas that should be stretched by drawing pixels along the left and top borders. You can also mark the area that should hold the content by drawing pixels along the right and bottom borders. The process is shown from Fig. 4.17 to Fig. 4.18.

Figure 4.17 Original image (.png).

The black pixels are along the borders. The ones on the top and

Figure 4.18 Nine-patch image (.9.png).

left borders indicate the places where the image can be stretched, and the ones on the right and bottom borders indicate where the content should be placed.

Figure 4.19 A nine-patch image used in various sizes.

4.2.2 Action Bar

The action bar, also called app bar, is one of the most important design elements in activities. It provides a visual structure and interactive elements that are familiar to users. A typical action bar is shown in Fig. 4.20.

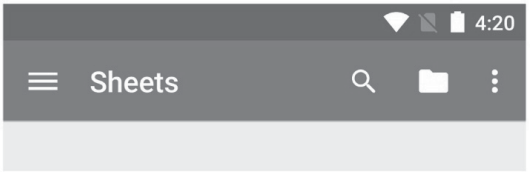

Figure 4.20 A typical action bar.

An action bar has some key functions listed as follows:

1. Dedicated space for giving the app an identity and indicating the user's virtual location in the app.
2. Access to important actions in a predictable way, such as search.
3. Support for navigation and view switching (with tabs or drop- down lists).

In its most basic form, the action bar displays the title for the activity on one side and an overflow menu on the other. Beginning with Android 3.0 (API level 11), all activities that use the default theme have an ActionBar as an app bar. However, app bar features have gradually been added to the native ActionBar over various Android releases. As a result, the native ActionBar behaves differently depending on what version of the Android system a device may be using. By contrast, the most recent features are added to the support library's version of Toolbar, and they are available on any device that can use the support library.

For this reason, you should use the support library's Toolbar class to implement your activities' app bars. Using the support library's toolbar helps ensure that your app will

have consistent behavior across the widest range of devices. For example, the Toolbar widget provides a material design experience on devices running Android 2.1 (API level 7) or later, but the native action bar doesn't support material design unless the device is running Android 5.0 (API level 21) or later.

4.2.3 Custom Views

Android has a large set of view classes for interacting with users and displaying various types of data. However sometimes we have some unique requirements that are not covered by the built-in views. To be a well-designed class, a custom view should:

1. conform to Android standards;
2. provide custom styleable attributes that work with Android XML layouts;
3. send accessibility events;
4. be compatible with multiple Android platforms.

All of the view classes defined in the Android framework extend the View. The custom view can also extend View directly, or we can extend some existing view subclasses, such as Button. Then we need to define some attributes for the custom view. To define custom attributes, add

<declare-styleable> resources to our project. It's customary to put these resources into a res/values/attrs.xml file.

After a view is created from an XML layout, all of the attributes in the XML tags are read from the resource bundle and passed into the view's constructor as an AttributeSet. Then we will pass the Attribute- Set to obtainStyledAttributes(). This method passes back a TypedAr- ray array of values that has already been dereferenced and styled.

Then we need to add properties and events to the custom view. To provide dynamic behavior, we need to expose a property getter and setter pair for each custom attribute,

for example, showing text and image. After creating and initiating the custom view, we move to the most important part of a custom view, which is its appearance. Furthermore, the most important step in drawing a custom view is to override the onDraw() method. The parameter to onDraw() is a Canvas object that the view can use to draw itself. The Canvas class defines methods for drawing text, lines, bitmaps, and many other graphics primitives. You can use these methods in onDraw() to create your custom UI.

Drawing a UI is only one part of creating a custom view. You also need to make your view respond to user input in a way that closely resembles the real-world action you're mimicking. We need to make the view interactive, including input gestures, physically plausible motion, and making transactions smooth.

4.3 OVERVIEW OF MULTIMEDIA IN ANDROID

4.3.1 Understanding the Media Player Class

Android support audio and video output through the Media Player class in the android.media package. The android.media is used to manage various media interfaces. The Media APIs are used to play and record media files, including audio and video . The Media Player class can be used to control playback of audio/video files and streams. The control of audio/video files and streams is managed as a state machine, as shown in Fig. 4.21.

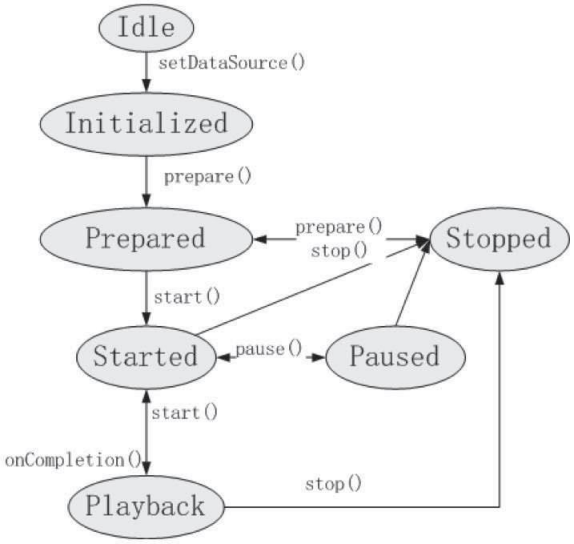

Figure 4.21 State diagram of the MediaPlayer.

4.3.2 Life Cycle of the MediaPlayer State

The life cycle and the state of a *MediaPlayer* object are driven by the supported playback control operations. The *setDataSource()* method is called to transfer a *MediaPlayer* object from the idle state to the initialized state. A *MediaPlayer* object must enter the prepared state first before it is started and played back. A *MediaPlayer* can enter the prepared state by call the *prepare()* or *prepareAsync()* method. The *prepare()* method transfers the object to the prepared state once the method call is returned. The *prepareAsync()* method first transfers the object to the preparing state after the call returns while the internal player engine continues working to complete the rest of the preparation work.

The *start()* method must be called to start the playback. The *Me- diaPlayer* object is in the started state, after *start()* returns. Calling *start()* has no effect on a *MediaPlayer* object that is already in the started state.