64. Implementing Video Playback on Android using the VideoView and MediaController Classes

One of the primary uses for smartphones and tablets is to enable the user to access and consume content. One key form of content widely used, especially in the case of tablet devices, is video.

The Android SDK includes two classes that make the implementation of video playback on Android devices extremely easy to implement when developing applications. This chapter will provide an overview of these two classes, VideoView and MediaController, before working through the creation of a simple video playback application.

64.1 Introducing the Android VideoView Class

By far the simplest way to display video within an Android application is to use the VideoView class. This is a visual component which, when added to the layout of an activity, provides a surface onto which a video may be played. Android currently supports the following video formats:

- H.263
- H.264 AVC
- H.265 HEVC
- MPEG-4 SP
- VP8
- VP9

The VideoView class has a wide range of methods that may be called in order to manage the playback of video. Some of the more commonly used methods are as follows:

- **setVideoPath(String path)** – Specifies the path (as a string) of the video media to be played. This can be either the URL of a remote video file or a video file local to the device.
- **setVideoUri(Uri uri)** – Performs the same task as the setVideoPath() method but takes a Uri object as an argument instead of a string.
- **start()** – Starts video playback.
- **stopPlayback()** – Stops the video playback.
- **pause()** – Pauses video playback.
- **isPlaying()** – Returns a Boolean value indicating whether a video is currently playing.
- **setOnPreparedListener(MediaPlayer.OnPreparedListener)** – Allows a callback method to be called when the video is ready to play.
- **setOnErrorListener(MediaPlayer.OnErrorListener)** - Allows a callback method to be called when an error occurs during the video playback.
- **setOnCompletionListener(MediaPlayer.OnCompletionListener)** - Allows a callback method to be called when the end of the video is reached.
- **getDuration()** – Returns the duration of the video. Will typically return -1 unless called from within the OnPreparedListener() callback method.
- **getCurrentPosition()** – Returns an integer value indicating the current position of playback.

- **setMediaController(MediaController)** – Designates a MediaController instance allowing playback controls to be displayed to the user.

64.2 Introducing the Android MediaController Class

If a video is simply played using the VideoView class, the user will not be given any control over the playback, which will run until the end of the video is reached. This issue can be addressed by attaching an instance of the MediaController class to the VideoView instance. The MediaController will then provide a set of controls allowing the user to manage the playback (such as pausing and seeking backwards/forwards in the video timeline).

The position of the controls is designated by anchoring the controller instance to a specific view in the user interface layout. Once attached and anchored, the controls will appear briefly when playback starts and may subsequently be restored at any point by the user tapping on the view to which the instance is anchored.

Some of the key methods of this class are as follows:

- **setAnchorView(View view)** – Designates the view to which the controller is to be anchored. This controls the location of the controls on the screen.
- **show()** – Displays the controls.
- **show(int timeout)** – Controls are displayed for the designated duration (in milliseconds).
- **hide()** – Hides the controller from the user.
- **isShowing()** – Returns a Boolean value indicating whether the controls are currently visible to the user.

64.3 Testing Video Playback

At the time of writing, it is not possible to test video playback when using the Android AVD emulators. To test the video playback functionality of an application it will be necessary to deploy it onto a physical device.

64.4 Creating the Video Playback Example

The remainder of this chapter is dedicated to working through an example application intended to use the VideoView and MediaController classes to play a web based MPEG-4 video file.

Create a new project in Android Studio, entering *VideoPlayer* 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 *VideoPlayerActivity* with a corresponding layout named *activity_video_player*.

64.5 Designing the VideoPlayer Layout

The user interface for the main activity will consist solely of an instance of the VideoView class. Use the Project tool window to locate the *app -> res -> layout -> activity_video_player.xml* file, double-click on it, switch the Layout Editor tool to Design mode and delete the default TextView widget.

From the Images category of the Palette panel, drag and drop a VideoView instance onto the layout so that it fills the available canvas area as shown in Figure 64-1.

Establish constraints from each side of the VideoView to the corresponding side of the parent layout. Use the Properties tool window to change the ID of the component to *videoView1* set the layout_width and layout_height properties for the VideoView instance to *match_parent*.

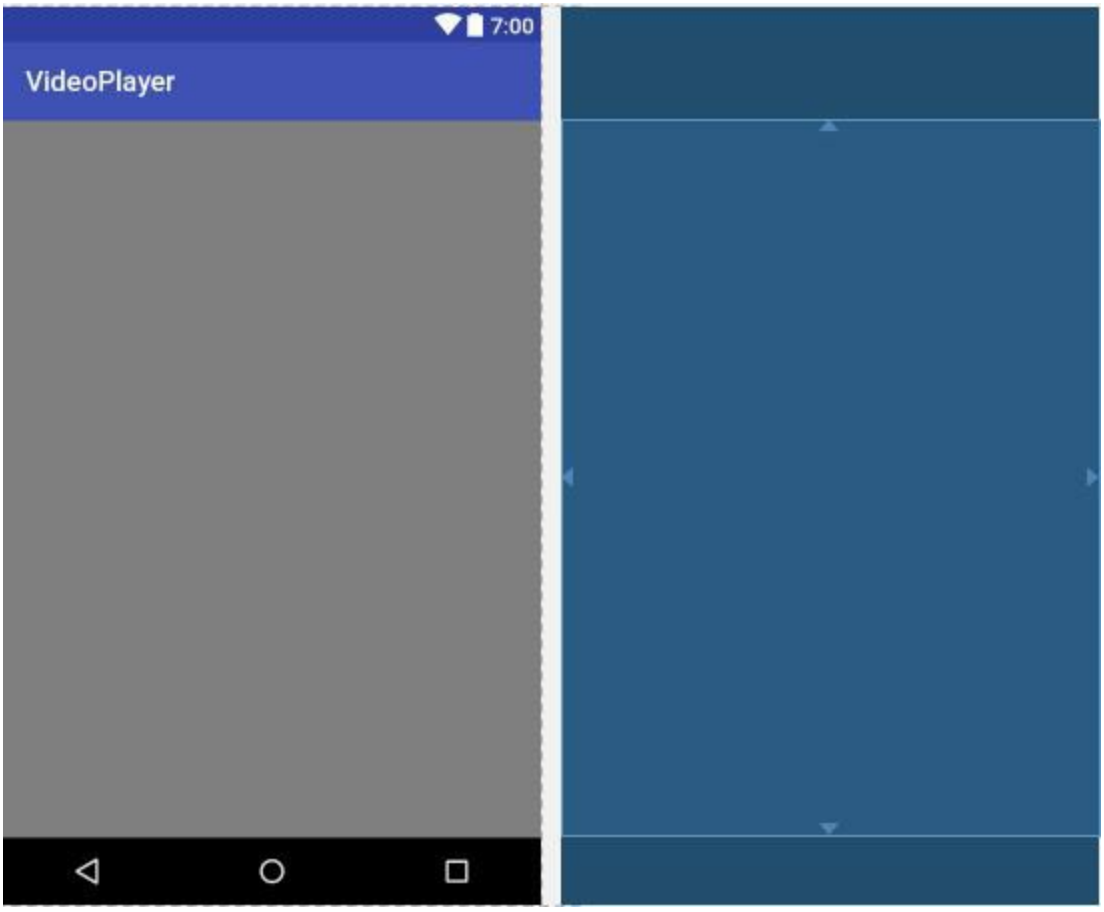

Figure 64-1

On completion of the layout design, the XML resources for the layout should read as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="com.ebookfrenzy.videoplayer.VideoPlayerActivity" >

    <VideoView
        android:id="@+id/videoView1"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

</android.support.constraint.ConstraintLayout>
```

64.6 Configuring the VideoView

The next step is to configure the VideoView with the path of the video to be played and then start the playback. This will be performed when the main activity has initialized, so load the *VideoPlayerActivity.java* file into the editor and modify the *OnCreate()* method as outlined in the following listing:

```
package com.ebookfrenzy.videoplayer;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.widget.VideoView;

public class VideoPlayerActivity extends AppCompatActivity {

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

        final VideoView videoView =
                (VideoView) findViewById(R.id.videoView1);

        videoView.setVideoPath(
                "http://www.ebookfrenzy.com/android_book/movie.mp4");

        videoView.start();
    }
}
```

All that this code does is obtain a reference to the VideoView instance in the layout, set the video path on it to point to an MPEG-4 file hosted on a web site and then start the video playing.

64.7 Adding Internet Permission

An attempt to run the application at this point would result in the application failing to launch with an error dialog appearing on the Android device that reads “Unable to Play Video. Sorry, this video cannot be played”. This is not because of an error in the code or an incorrect video file format. The issue would be that the application is attempting to access a file over the internet, but has failed to request appropriate permissions to do so. To resolve this, edit the *AndroidManifest.xml* file for the project and add a line to request internet access:

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

    <uses-permission android:name="android.permission.INTERNET" />

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:supportsRtl="true"
        android:theme="@style/AppTheme" >
.
.
.
```

```
</manifest>
```

Test the application by running it on a physical Android device. After the application launches there may be a short delay while video content is buffered before the playback begins (Figure 64-2).

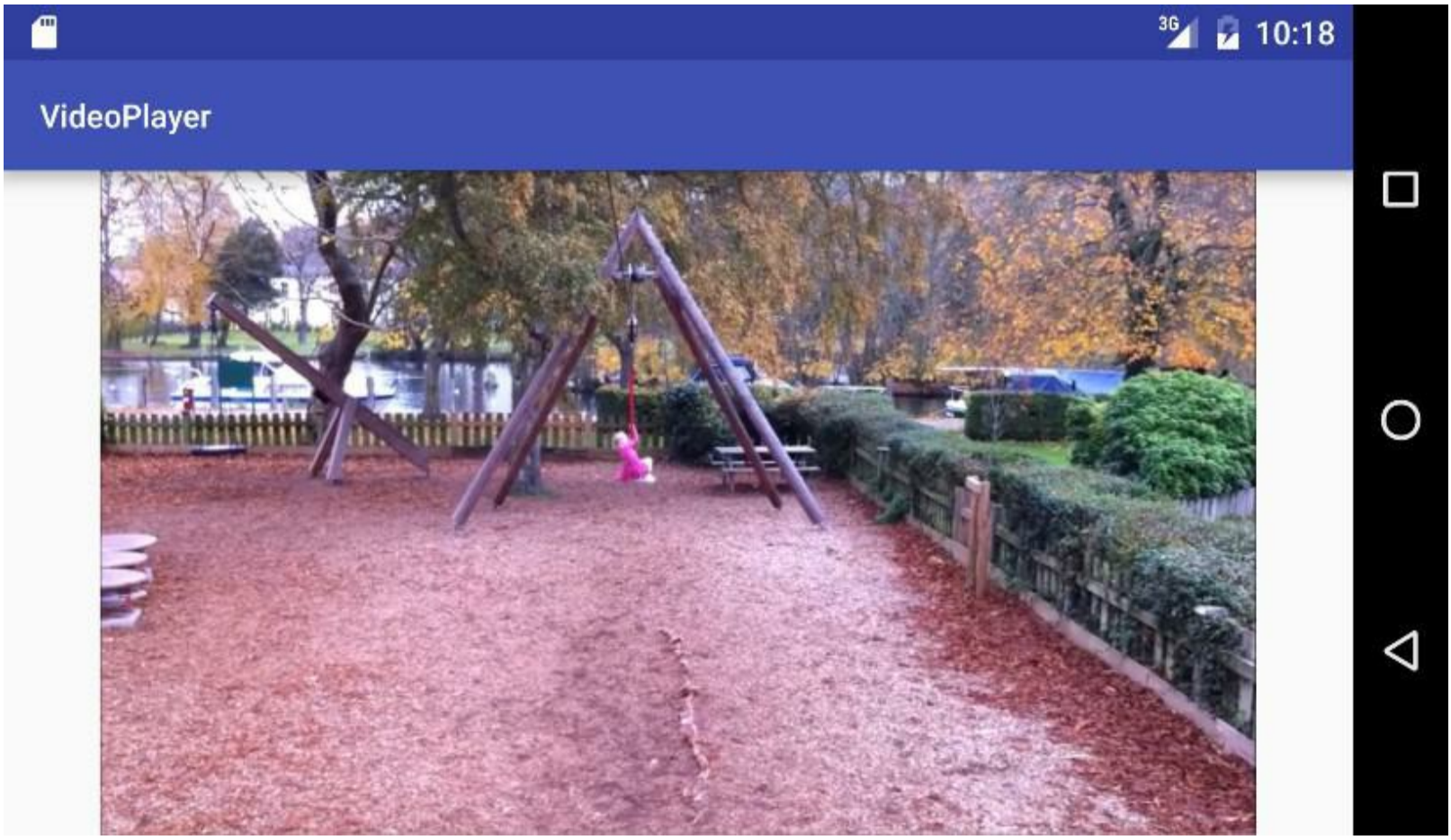

Figure 64-2

This provides an indication of how easy it can be to integrate video playback into an Android application. Everything so far in this example has been achieved using a VideoView instance and three lines of code.

64.8 Adding the MediaController to the Video View

As the VideoPlayer application currently stands, there is no way for the user to control playback. As previously outlined, this can be achieved using the MediaController class. To add a controller to the VideoView, modify the *onCreate()* method once again:

```
package com.ebookfrenzy.videoplayer;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.widget.VideoView;
import android.widget.MediaController;

public class VideoPlayerActivity extends AppCompatActivity {

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

```
                final VideoView videoView = (VideoView)
                                findViewById(R.id.videoView1);

                videoView.setVideoPath(
                   "http://www.ebookfrenzy.com/android_book/movie.mp4");

                MediaController mediaController = new
                        MediaController(this);
                mediaController.setAnchorView(videoView);
                videoView.setMediaController(mediaController);

                videoView.start();
        }
}
```

When the application is launched with these changes implemented, tapping the VideoView canvas will cause the media controls will appear over the video playback. These controls should include a seekbar together with fast forward, rewind and play/pause buttons. After the controls recede from view, they can be restored at any time by tapping on the VideoView canvas once again. With just three more lines of code, our video player application now has media controls as shown in Figure 64-3:

Figure 64-3

64.9 Setting up the onPreparedListener

As a final example of working with video based media, the *onCreate()* method will now be extended further to demonstrate the mechanism for configuring a listener. In this case, a listener will be implemented that is intended to output the duration of the video as a message in the Android Studio LogCat panel:

```
package com.ebookfrenzy.videoplayer;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.widget.VideoView;
import android.widget.MediaController;
import android.util.Log;
import android.media.MediaPlayer;

public class VideoPlayerActivity extends AppCompatActivity {

    String TAG = "VideoPlayer";

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

```
        final VideoView videoView =
                (VideoView) findViewById(R.id.videoView1);

        videoView.setVideoPath(
                "http://www.ebookfrenzy.com/android_book/movie.mp4");

        MediaController mediaController = new
                MediaController(this);
        mediaController.setAnchorView(videoView);
        videoView.setMediaController(mediaController);

        videoView.setOnPreparedListener(new
                MediaPlayer.OnPreparedListener()  {
                    @Override
                    public void onPrepared(MediaPlayer mp) {
                            Log.i(TAG, "Duration = " +
                                    videoView.getDuration());
                    }
         });

        videoView.start();

    }
}
```

Now just before the video playback begins, a message will appear in the Android Studio LogCat panel that reads along the lines of:

```
11-05 10:27:52.256 12542-12542/com.ebookfrenzy.videoplayer I/VideoPlayer:
Duration = 6874
```

64.10 **Summary**

Tablet based Android devices make excellent platforms for the delivery of content to users, particularly in the form of video media. As outlined in this chapter, the Android SDK provides two classes, namely VideoView and MediaController, which combine to make the integration of video playback into Android applications quick and easy, often involving just a few lines of Java code.

65. Video Recording and Image Capture on Android using Camera Intents

Many Android devices are equipped with at least one camera. There are a number of ways to allow the user to record video from within an Android application via these built-in cameras, but by far the easiest approach is to make use of a camera intent included with the Android operating system. This allows an application to invoke the standard Android video recording interface. When the user has finished recording, the intent will return to the application, passing through a reference to the media file containing the recorded video.

As will be demonstrated in this chapter, this approach allows video recording capabilities to be added to applications with just a few lines of code.

65.1 Checking for Camera Support

Before attempting to access the camera on an Android device, it is essential that defensive code be implemented to verify the presence of camera hardware. This is of particular importance since not all Android devices include a camera.

The presence or otherwise of a camera can be identified via a call to the *PackageManager.hasSystemFeature()* method. In order to check for the presence of a front-facing camera, the code needs to check for the presence of the *PackageManager.FEATURE_CAMERA_FRONT* feature. This can be encapsulated into the following convenience method:

```
private boolean hasCamera() {
       return (getPackageManager().hasSystemFeature(
           PackageManager.FEATURE_CAMERA_FRONT));
}
```

The presence of a camera facing away from the device screen can be similarly verified using the *PackageManager.FEATURE_CAMERA* constant. A test for whether a device has any camera can be performed by referencing *PackageManager.FEATURE_CAMERA_ANY*.

65.2 Calling the Video Capture Intent

Use of the video capture intent involves, at a minimum, the implementation of code to call the intent activity and a method to handle the return from the activity. The Android built-in video recording intent is represented by *MediaStore.ACTION_VIDEO_CAPTURE* and may be launched as follows:

```
private static final int VIDEO_CAPTURE = 101;

Intent intent = new Intent(MediaStore.ACTION_VIDEO_CAPTURE);
startActivityForResult(intent, VIDEO_CAPTURE);
```

When invoked in this way, the intent will place the recorded video into a file using a default location and file name.

When the user either completes or cancels the video recording session, the *onActivityResult()* method of the calling activity will be called. This method needs to check that the request code passed through as an argument matches that specified when the intent was launched, verify that the recording session

was successful and extract the path of the video media file. The corresponding *onActivityResult()* method for the above intent launch code might, therefore, be implemented as follows:

```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent
data) {
    Uri videoUri = data.getData();

    if (requestCode == VIDEO_CAPTURE) {
        if (resultCode == RESULT_OK) {
            Toast.makeText(this, "Video saved to:\n" +
                    videoUri, Toast.LENGTH_LONG).show();
        } else if (resultCode == RESULT_CANCELED) {
            Toast.makeText(this, "Video recording cancelled.",
                    Toast.LENGTH_LONG).show();
        } else {
            Toast.makeText(this, "Failed to record video",
                    Toast.LENGTH_LONG).show();
        }
    }

}
```

The above code example simply displays a toast message indicating the success of the recording intent session. In the event of a successful recording, the path to the stored video file is displayed.

When executed, the video capture intent (Figure 65-1) will launch and provide the user the opportunity to record video.

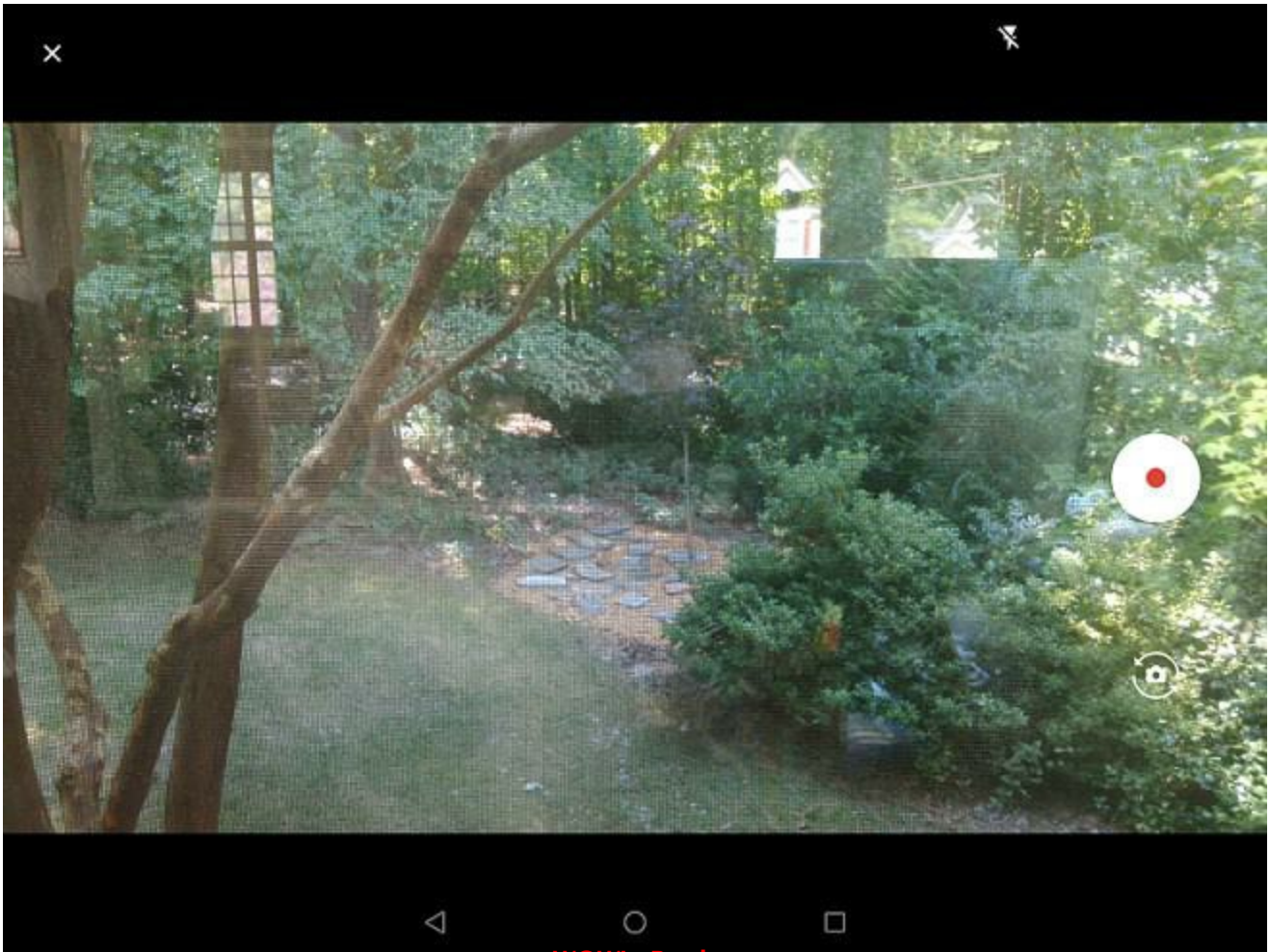

Figure 65-1

65.3 Calling the Image Capture Intent

In addition to the video capture intent, Android also includes an intent designed for taking still photos using the built-in camera, launched by referencing *MediaStore.ACTION_IMAGE_CAPTURE*:

```
private static final int IMAGE_CAPTURE = 102;

Intent intent = new Intent(MediaStore.ACTION_IMAGE_CAPTURE);
startActivityForResult(intent, IMAGE_CAPTURE);
```

As with video capture, the intent may be passed the location and file name into which the image is to be stored, or left to use the default location and naming convention.

65.4 Creating an Android Studio Video Recording Project

In the remainder of this chapter, a very simple application will be created to demonstrate the use of the video capture intent. The application will consist of a single button which will launch the video capture intent. Once video has been recorded and the video capture intent dismissed, the application will simply display the path to the video file as a Toast message. The VideoPlayer application created in the previous chapter may then be modified to play back the recorded video.

Create a new project in Android Studio, entering *CameraApp* 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 *CameraAppActivity* with a layout file named *activity_camera_app*.

65.5 Designing the User Interface Layout

Navigate to *app -> res -> layout* and double-click on the *activity_camera_app.xml* layout file to load it into the Layout Editor tool.

With the Layout Editor tool in Design mode, delete the default “Hello World!” text view and replace it with a Button view positioned in the center of the layout canvas. Change the text on the button to read “Record Video” and extract the text to a string resource. Also, assign an *onClick* property to the button so that it calls a method named *startRecording* when selected by the user and change the layout_width property to *wrap_content*:

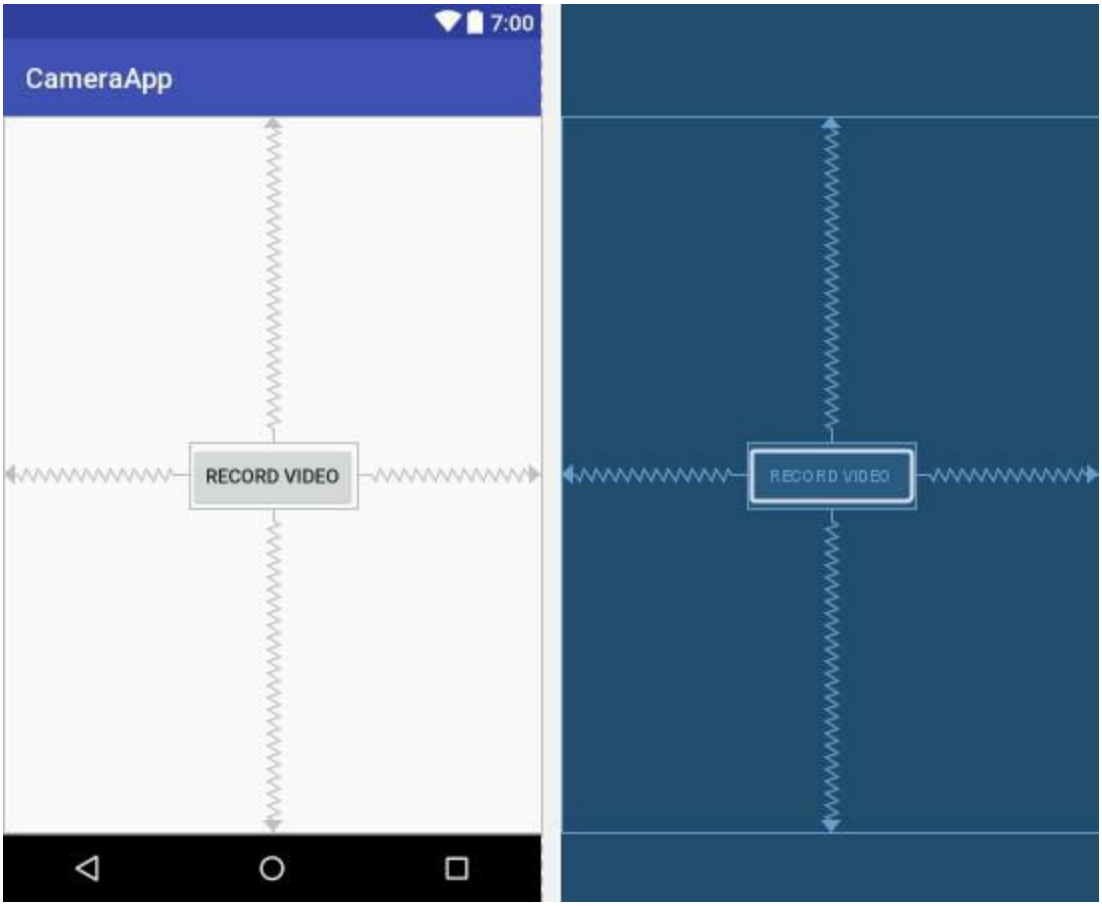

Figure 65-2

Remaining within the Properties tool window, change the ID to *recordButton*.

65.6 Checking for the Camera

Before attempting to launch the video capture intent, the application first needs to verify that the device on which it is running actually has a camera. For the purposes of this example, we will simply make use of the previously outlined *hasCamera()* method, this time checking for any camera type. In the event that a camera is not present, the Record Video button will be disabled.

Edit the *CameraAppActivity.java* file and modify it as follows:

```
package com.ebookfrenzy.cameraapp;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.content.pm.PackageManager;
import android.widget.Button;

public class CameraAppActivity extends AppCompatActivity {

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

                Button recordButton =
                  (Button) findViewById(R.id.recordButton);

                if (!hasCamera())
                        recordButton.setEnabled(false);
```

```
        }

        private boolean hasCamera() {
             return (getPackageManager().hasSystemFeature(
                                 PackageManager.FEATURE_CAMERA_ANY));
        }
    }
```

65.7 Launching the Video Capture Intent

The objective is for the video capture intent to launch when the user selects the *Record Video* button. Since this is now configured to call a method named *startRecording()*, the next logical step is to implement this method within the *CameraAppActivity.java* source file:

```
package com.ebookfrenzy.cameraapp;

import java.io.File;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.content.pm.PackageManager;
import android.widget.Button;
import android.net.Uri;
import android.os.Environment;
import android.provider.MediaStore;
import android.content.Intent;
import android.view.View;

public class CameraAppActivity extends AppCompatActivity {

    private static final int VIDEO_CAPTURE = 101;
    private Uri fileUri;

    public void startRecording(View view)
    {
         Intent intent = new Intent(MediaStore.ACTION_VIDEO_CAPTURE);
         startActivityForResult(intent, VIDEO_CAPTURE);
    }
.
.
.
}
```

65.8 Handling the Intent Return

When control returns back from the intent to the application's main activity, the *onActivityResult()* method will be called. All that this method needs to do for this example is verify the success of the video capture and display the path of the file into which the video has been stored:

```
package com.ebookfrenzy.cameraapp;

import java.io.File;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.content.pm.PackageManager;
```

```
import android.widget.Button;
import android.net.Uri;
import android.os.Environment;
import android.provider.MediaStore;
import android.content.Intent;
import android.view.View;
import android.widget.Toast;

public class CameraAppActivity extends AppCompatActivity {
.
.
.
    protected void onActivityResult(int requestCode,
              int resultCode, Intent data) {

        Uri videoUri = data.getData();

        if (requestCode == VIDEO_CAPTURE) {
            if (resultCode == RESULT_OK) {
                Toast.makeText(this, "Video saved to:\n" +
                        videoUri, Toast.LENGTH_LONG).show();
            } else if (resultCode == RESULT_CANCELED) {
                Toast.makeText(this, "Video recording cancelled.",
                        Toast.LENGTH_LONG).show();
            } else {
                Toast.makeText(this, "Failed to record video",
                        Toast.LENGTH_LONG).show();
            }
    }
.
.
}
```

65.9 Testing the Application

Compile and run the application on a physical Android device or emulator session, touch the record button and use the video capture intent to record some video. Once completed, stop the video recording. Play back the recording by selecting the play button on the screen. Finally, touch the *Done* (sometimes represented by a check mark) button on the screen to return to the CameraApp application. On returning, a Toast message should appear stating that the video has been stored in a specific location on the device (the exact location will differ from one device type to another) from where it can be moved, stored or played back depending on the requirements of the app.

65.10 Summary

Most Android tablet and smartphone devices include a camera that can be accessed by applications. While there are a number of different approaches to adding camera support to applications, the Android video and image capture intents provide a simple and easy solution to capturing video and images.