CHAPTER 8

Using the Camera

Most Android devices will have a camera, since they are fairly commonplace on mobile devices these days. You, as an Android developer, can take advantage of the camera, for everything from snapping tourist photos to scanning barcodes. For simple operations, the APIs needed to use the camera are fairly straight-forward, requiring a bit of boilerplate code plus your own unique application logic.

What is a problem is using the camera with the emulator. The emulator does not emulate a camera, nor is there a convenient way to pretend there are pictures via DDMS or similar tools. For the purposes of this chapter, it is assumed you have access to an actual Android-powered hardware device and can use it for development purposes.

Sneaking a Peek

First, it is fairly common for a camera-using application to support a preview mode, to show the user what the camera sees. This will help make sure the camera is lined up on the subject properly, whether there is sufficient lighting, etc.

So, let us take a look at how to create an application that shows such a live preview. The code snippets shown in this section are pulled from the `Camera/Preview` sample project.

The Permission

First, you need permission to use the camera. That way, when end users install your application off of the Internet, they will be notified that you intend to use the camera, so they can determine if they deem that appropriate for your application.

You simply need the `CAMERA` permission in your `AndroidManifest.xml` file, along with whatever other permissions your application logic might require. Here is the manifest from the `Camera/Preview` sample project:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.commonsware.android.camera"
      android:versionCode="1"
      android:versionName="1.0">
  <uses-permission android:name="android.permission.CAMERA" />
    <application android:label="@string/app_name">
        <activity android:name=".PreviewDemo"
                  android:label="@string/app_name"
                  android:configChanges="keyboardHidden|orientation"
                  android:screenOrientation="landscape"
                  android:theme="@android:style/Theme.NoTitleBar.Fullscreen">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

Also note a few other things about our `PreviewDemo` activity as registered in this manifest:

- We use `android:configChanges = "keyboardHidden|orientation"` to ensure we control what happens when the keyboard is hidden or exposed, rather than have Android rotate the screen for us
- We use `android:screenOrientation = "landscape"` to tell Android we are always in landscape mode. This is necessary because of a bit of a bug in the camera preview logic, such that it works best in landscape mode.

- We use `android:theme = "@android:style/Fullscreen"` to get rid of the title bar and status bar, so the preview is truly full-screen (e.g., 480x320 on a T-Mobile G1).

The SurfaceView

Next, you need a layout supporting a `SurfaceView`. `SurfaceView` is used as a raw canvas for displaying all sorts of graphics outside of the realm of your ordinary widgets. In this case, Android knows how to display a live look at what the camera sees on a `SurfaceView`, to serve as a preview pane.

For example, here is a full-screen `SurfaceView` layout as used by the `PreviewDemo` activity:

```
<?xml version="1.0" encoding="utf-8"?>
<android.view.SurfaceView
xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/preview"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
</android.view.SurfaceView>
```

The Camera

The biggest step, of course, is telling Android to use the camera service and tie a camera to the `SurfaceView` to show the actual preview. We will also eventually need the camera service to take real pictures, as will be described in the next section.

There are three major components to getting picture preview working:

1. The `SurfaceView`, as defined in our layout
2. A `SurfaceHolder`, which is a means of controlling behavior of the `SurfaceView`, such as its size, or being notified when the surface changes, such as when the preview is started
3. A `Camera`, obtained from the `open()` static method on the `Camera` class

To wire these together, we first need to:

- Get the `SurfaceHolder` for our `SurfaceView` via `getHolder()`
- Register a `SurfaceHolder.Callback` with the `SurfaceHolder`, so we are notified when the `SurfaceView` is ready or changes
- Tell the `SurfaceView` (via the `SurfaceHolder`) that it has the `SURFACE_TYPE_PUSH_BUFFERS` type (`setType()`) – this indicates something in the system will be updating the `SurfaceView` and providing the bitmap data to display

This gives us a configured `SurfaceView` (shown below), but we still need to tie in the `Camera`.

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

  preview=(SurfaceView)findViewById(R.id.preview);
  previewHolder=preview.getHolder();
  previewHolder.addCallback(surfaceCallback);
  previewHolder.setType(SurfaceHolder.SURFACE_TYPE_PUSH_BUFFERS);
}
```

A `Camera` object has a `setPreviewDisplay()` method that takes a `SurfaceHolder` and, as you might expect, arranges for the camera preview to be displayed on the associated `SurfaceView`. However, the `SurfaceView` may not be ready immediately after being changed into `SURFACE_TYPE_PUSH_BUFFERS` mode. So, while the previous setup work could be done in `onCreate()`, you should wait until the `SurfaceHolder.Callback` has its `surfaceCreated()` method called, then register the `Camera`:

```
public void surfaceCreated(SurfaceHolder holder) {
  camera=Camera.open();

  try {
    camera.setPreviewDisplay(previewHolder);
  }
  catch (Throwable t) {
    Log.e("PreviewDemo-surfaceCallback",
          "Exception in setPreviewDisplay()", t);
    Toast
      .makeText(PreviewDemo.this, t.getMessage(), Toast.LENGTH_LONG)
      .show();
```

```
  }
}
```

Next, once the `SurfaceView` is set up and sized by Android, we need to pass configuration data to the `Camera`, so it knows how big to draw the preview. Since the preview pane is not a fixed size – it might vary based on hardware – we cannot safely pre-determine the size. It is simplest to wait for our `SurfaceHolder.Callback` to have its `surfaceChanged()` method called, when we are told the size of the surface. Then, we can pour that information into a `Camera.Parameters` object, update the `Camera` with those parameters, and have the `Camera` show the preview images via `startPreview()`:

```
public void surfaceChanged(SurfaceHolder holder,
                           int format, int width,
                           int height) {
  Camera.Parameters parameters=camera.getParameters();

  parameters.setPreviewSize(width, height);
  camera.setParameters(parameters);
  camera.startPreview();
}
```

Eventually, the preview needs to stop. In this particular case, that will be as the activity is being destroyed. It is important to release the `Camera` at this time – for many devices, there is only one physical camera, so only one activity can be using it at a time. Our `SurfaceHolder.Callback` will be told, via `surfaceDestroyed()`, when it is being closed up, and we can stop the preview (`stopPreview()`), release the camera (`release()`), and let go of it (`camera = null`) at that point:

```
public void surfaceDestroyed(SurfaceHolder holder) {
  camera.stopPreview();
  camera.release();
  camera=null;
}
```

If you compile and run the `Camera/Preview` sample application, you will see, on-screen, what the camera sees.

Here is the full `SurfaceHolder.Callback` implementation:

```
SurfaceHolder.Callback surfaceCallback=new SurfaceHolder.Callback() {
  public void surfaceCreated(SurfaceHolder holder) {
    camera=Camera.open();

    try {
      camera.setPreviewDisplay(previewHolder);
    }
    catch (Throwable t) {
      Log.e("PreviewDemo-surfaceCallback",
            "Exception in setPreviewDisplay()", t);
      Toast
        .makeText(PreviewDemo.this, t.getMessage(), Toast.LENGTH_LONG)
        .show();
    }
  }

  public void surfaceChanged(SurfaceHolder holder,
                             int format, int width,
                             int height) {
    Camera.Parameters parameters=camera.getParameters();

    parameters.setPreviewSize(width, height);
    camera.setParameters(parameters);
    camera.startPreview();
  }

  public void surfaceDestroyed(SurfaceHolder holder) {
    camera.stopPreview();
    camera.release();
    camera=null;
  }
};
```

Image Is Everything

Showing the preview imagery is nice and all, but it is probably more important to actually take a picture now and again. The previews show the user what the camera sees, but we still need to let our application know what the camera sees at particular points in time.

In principle, this is easy. Where things get a bit complicated comes with ensuring the application (and device as a whole) has decent performance, not slowing down to process the pictures.

The code snippets shown in this section are pulled from the `Camera/Picture` sample project, which builds upon the `Camera/Preview` sample shown in the previous section.

Asking for a Format

We need to tell the `Camera` what sort of picture to take when we decide to take a picture. The two options are raw and JPEG.

At least, that is the theory.

In practice, the T-Mobile G1 does not support raw output, only JPEG. So, we need to tell the `Camera` that we want JPEG output.

That is merely a matter of calling `setPictureFormat()` on the `Camera.Parameters` object when we configure our `Camera`, using the value `JPEG` to indicate that we, indeed, want JPEG:

```
public void surfaceChanged(SurfaceHolder holder,
                           int format, int width,
                           int height) {
  Camera.Parameters parameters=camera.getParameters();

  parameters.setPreviewSize(width, height);
  parameters.setPictureFormat(PixelFormat.JPEG);
  camera.setParameters(parameters);
  camera.startPreview();
}
```

Connecting the Camera Button

Somehow, your application will need to indicate when a picture should be taken. That could be via widgets on the UI, though in our samples here, the preview is full-screen.

An alternative is to use the camera hardware button. Like every hardware button other than the Home button, we can find out when the camera button is clicked via `onKeyDown()`:

```
@Override
public boolean onKeyDown(int keyCode, KeyEvent event) {
  if (keyCode==KeyEvent.KEYCODE_CAMERA ||
      keyCode==KeyEvent.KEYCODE_SEARCH) {
    takePicture();

    return(true);
  }

  return(super.onKeyDown(keyCode, event));
}
```

Since the HTC Magic does not have a hardware camera button, we also watch for `KEYCODE_SEARCH` for the dedicated search key, which is in the upper-right portion of the Magic's face when the device is held in landscape mode.

Taking a Picture

Once it is time to take a picture, all you need to do is:

- Stop the preview
- Tell the `Camera` to `takePicture()`

The `takePicture()` method takes three parameters, all callback-style objects:

1. A "shutter" callback (`Camera.ShutterCallback`), which is notified when the picture has been captured by the hardware but the data is not yet available – you might use this to play a "camera click" sound
2. Callbacks to receive the image data, either in raw format or JPEG format

Since the T-Mobile G1 only supports JPEG output, and because we do not want to fuss with a shutter click, `PictureDemo` only passes in the third parameter to `takePicture()`:

```
private void takePicture() {
  camera.stopPreview();
  camera.takePicture(null, null, photoCallback);
}
```

The `Camera.PictureCallback` (`photoCallback`) needs to implement `onPictureTaken()`, which provides the picture data as a `byte[]`, plus the `Camera` object that took the picture. At this point, it is safe to start up the preview again.

Plus, of course, it would be nice to do something with that byte array.

The catch is that the byte array is going to be large – the T-Mobile G1 has a 3-megapixel camera, and future hardware is more likely to have richer hardware than that. Writing that to flash, or sending it over the network, or doing just about anything with the data, will be slow. Slow is fine...so long as it is not on the UI thread.

That means we need to do a little more work.

Using AsyncTask

In theory, we could just fork a background thread to save off the image data or do whatever it is we wanted done with it. However, we could wind up with several such threads, particularly if we are sending the image over the Internet and do not have a fast connection to our destination server.

Android 1.5 offers a work queue model, in the form of `AsyncTask`. `AsyncTask` manages a thread pool and work queue – all we need to do is hand it the work to be done.

So, we can create an `AsyncTask` implementation, called `SavePhotoTask`, as follows:

```
class SavePhotoTask extends AsyncTask<byte[], String, String> {
  @Override
  protected String doInBackground(byte[]... jpeg) {
    File photo=new File(Environment.getExternalStorageDirectory(),
                        "photo.jpg");

    if (photo.exists()) {
      photo.delete();
    }
```

```
    try {
      FileOutputStream fos=new FileOutputStream(photo.getPath());

      fos.write(jpeg[0]);
      fos.close();
    }
    catch (java.io.IOException e) {
      Log.e("PictureDemo", "Exception in photoCallback", e);
    }

    return(null);
  }
}
```

Our `doInBackground()` implementation gets the byte array we received from Android. The byte array is simply the JPEG itself, so the data could be written to a file, transformed, sent to a Web service, converted into a `BitmapDrawable` for display on the screen or whatever.

In the case of `PictureDemo`, we take the simple approach of writing the JPEG file as `photo.jpg` in the root of the SD card. The byte array itself will be garbage collected once we are done saving it, so there is no explicit "free" operation we need to do to release that memory.

Finally, we arrange for our `PhotoCallback` to execute our `SavePhotoTask`:

```
Camera.PictureCallback photoCallback=new Camera.PictureCallback() {
  public void onPictureTaken(byte[] data, Camera camera) {
    new SavePhotoTask().execute(data);
    camera.startPreview();
  }
};
```