Chapter 5

An Android Game Development Framework

We've been through four chapters already and haven't written a single line of game code. The reason I've put you through all this boring theory and let you implement silly little test programs is simple: if you want to write games, you have to know exactly what's going on. You can't just copy and paste together code from all over the Web and hope that it will magically form the next first-person shooter hit. You should now have a firm grasp on how to design a simple game from the ground up, how to structure a nice API for 2D game development, and which Android APIs provide the functionality to implement your ideas.

To make Mr. Nom a reality, we have to do two things: implement the game framework interfaces and classes we designed in Chapter 3, and based on that, code up the game mechanics of Mr. Nom. Let's start with the game framework by merging what we designed in Chapter 3 with what we discussed in Chapter 4. Ninety percent of the code should be familiar to you already, since we did most of it in the tests in the last chapter.

Plan of Attack

In Chapter 3 we laid out a very minimal and clean design for a game framework that abstracts away all the platform specifics and let's us concentrate on what we are here for: game development. We'll implement all these interfaces and abstract classes now, in a bottom-up fashion, from easiest to hardest. The interfaces of Chapter 3 are located in the package `com.badlogic.androidgames.framework`. We'll put our implementation in the package `com.badlogic.androidgames.framework.impl`, indicating that this holds the actual implementation of the framework for Android. We'll prefix all our interface implementations with `Android` so that we can distinguish them from the interfaces. Let's start off with the easiest part: file I/O.

The code of this and the next chapter will be merged into a single Eclipse project. For now, just create a new Android project in Eclipse following the steps in the last chapter. How you name your default activity at this point doesn't matter for now.

The AndroidFileIO Class

The original `FileIO` interface was lean and mean. It only contained three methods: one to get an `InputStream` for an asset, another to get an `InputStream` for a file on the external storage, and a third that returns an `OutputStream` for a file on the external storage. In Chapter 4 you learned how we can open assets and files on the external storage with the Android APIs. Listing 5–1 shows you the implementation of the `FileIO` interface we'll use based on the knowledge from Chapter 4.

Listing 5–1. *AndroidFileIO.java; Implementing the FileIO Interface*

```
package com.badlogic.androidgames.framework.impl;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

import android.content.res.AssetManager;
import android.os.Environment;

import com.badlogic.androidgames.framework.FileIO;

public class AndroidFileIO implements FileIO {
    AssetManager assets;
    String externalStoragePath;

    public AndroidFileIO(AssetManager assets) {
        this.assets = assets;
        this.externalStoragePath = Environment.getExternalStorageDirectory()
                .getAbsolutePath() + File.separator;
    }

    @Override
    public InputStream readAsset(String fileName) throws IOException {
        return assets.open(fileName);
    }

    @Override
    public InputStream readFile(String fileName) throws IOException {
        return new FileInputStream(externalStoragePath + fileName);
    }

    @Override
    public OutputStream writeFile(String fileName) throws IOException {
        return new FileOutputStream(externalStoragePath + fileName);
    }
}
```

Everything's straightforward. We implement the FileIO interface, store an AssetManager along with the path of the external storage, and implement the three methods based on this. We pass through any IOExceptions that get thrown so we'll know if anything is fishy on the calling side.

Our Game interface implementation will hold an instance of this class and return it via Game.getFileIO(). This also means that our Game implementation will need to pass in the AssetManager later on for the AndroidFileIO instance to work.

Note that we do not check for the external storage to be available. If it's not available, or if we forgot to add the proper permission to the manifest file, we'll get an exception, so error checking is done implicitly. So we can move on to the next pieces of our framework: audio.

AndroidAudio, AndroidSound, and AndroidMusic: Crash, Bang, Boom!

We designed three interfaces in Chapter 3 for all our audio needs: Audio, Sound, and Music. Audio is responsible for creating Sound and Music instances from asset files. Sound let's us playback sound effects completely stored in RAM, and Music streams bigger music files from disk to the audio card. In Chapter 4 you learned what Android APIs we need to implement this. We start off with the implementation of AndroidAudio, as shown in Listing 5–2.

Listing 5–2. ***AndroidAudio.java; Implementing the Audio Interface***

```
package com.badlogic.androidgames.framework.impl;

import java.io.IOException;

import android.app.Activity;
import android.content.res.AssetFileDescriptor;
import android.content.res.AssetManager;
import android.media.AudioManager;
import android.media.SoundPool;

import com.badlogic.androidgames.framework.Audio;
import com.badlogic.androidgames.framework.Music;
import com.badlogic.androidgames.framework.Sound;

public class AndroidAudio implements Audio {
    AssetManager assets;
    SoundPool soundPool;
```

The AndroidAudio implementation has an AssetManager and a SoundPool instance. The AssetManager is needed so that we can load sound effects from asset files into the SoundPool on a call to AndroidAudio.newSound(). The SoundPool itself is also managed by the AndroidAudio instance.

```
    public AndroidAudio(Activity activity) {
        activity.setVolumeControlStream(AudioManager.STREAM_MUSIC);
        this.assets = activity.getAssets();
        this.soundPool = new SoundPool(20, AudioManager.STREAM_MUSIC, 0);
    }
```

In the constructor we pass in the Activity of our game for two reasons: it allows us to set the volume control to the media stream (remember we always want to do that), and it gives us an AssetManager instance, which we happily store in the corresponding member of the class. The SoundPool is configured to be able to play back 20 sound effects in parallel—enough for our needs.

```
    @Override
    public Music newMusic(String filename) {
        try {
            AssetFileDescriptor assetDescriptor = assets.openFd(filename);
            return new AndroidMusic(assetDescriptor);
        } catch (IOException e) {
            throw new RuntimeException("Couldn't load music '" + filename + "'");
        }
    }
```

The newMusic() method creates a new AndroidMusic instance. The constructor of that class takes an AssetFileDescriptor, from which it creates a MediaPlayer internally (more on that in a bit). The AssetManager.openFd() method throws an IOException in case something goes wrong. We catch it and rethrow it as a RuntimeException. Why not hand the IOException to the caller? First, it would clutter the calling code considerably, so we would rather throw a RuntimeException, which does not have to be caught explicitly. Second, we load the music from an asset file. It will only fail if we actually forget to add the music file to the assets/ directory or if our music file contains bogus bytes. These would constitute unrecoverable errors, as we need that Music instance for our game to function properly. To avoid that, we'll employ the strategy of throwing a RuntimeException instead of checked exceptions in a few more places in our game framework.

```
    @Override
    public Sound newSound(String filename) {
        try {
            AssetFileDescriptor assetDescriptor = assets.openFd(filename);
            int soundId = soundPool.load(assetDescriptor, 0);
            return new AndroidSound(soundPool, soundId);
        } catch (IOException e) {
            throw new RuntimeException("Couldn't load sound '" + filename + "'");
        }
    }
}
```

Finally, the newSound() method loads a sound effect from an asset into the SoundPool and returns an AndroidSound instance. The constructor of that instance takes a SoundPool and the ID of the sound effect the SoundPool assigned to it. We again throw any checked exception and rethrow it as an unchecked RuntimeException.

NOTE: We do not release the SoundPool in any of the methods. The reason for this is that there will always be a single Game instance holding a single Audio instance that holds a single SoundPool instance. The SoundPool instance will thus be alive as long as the activity (and with it our game) is alive. It will be destroyed automatically as soon as the activity drops dead.

Next up is the AndroidSound class, which implements the Sound interface. Listing 5–3 shows you its implementation.

Listing 5–3. *AndroidSound.java; Implementing the Sound Interface*

```
package com.badlogic.androidgames.framework.impl;

import android.media.SoundPool;

import com.badlogic.androidgames.framework.Sound;

public class AndroidSound implements Sound {
    int soundId;
    SoundPool soundPool;

    public AndroidSound(SoundPool soundPool, int soundId) {
        this.soundId = soundId;
        this.soundPool = soundPool;
    }

    @Override
    public void play(float volume) {
        soundPool.play(soundId, volume, volume, 0, 0, 1);
    }

    @Override
    public void dispose() {
        soundPool.unload(soundId);
    }
}
```

No surprises here. We simply store the SoundPool and the ID of the loaded sound effect for later playback and disposal via the play() and dispose() methods. It doesn't get any easier. All hail to the Android API.

Finally we have to implement the AndroidMusic class returned by AndroidAudio.newMusic(). Listing 5–4 shows the code for that class. It looks a little more complex than before. That's due to the state machine that the MediaPlayer really uses, which will throw exceptions like mad if we call methods in certain states.

Listing 5–4. *AndroidMusic.java; Implementing the Music Interface*

```
package com.badlogic.androidgames.framework.impl;

import java.io.IOException;

import android.content.res.AssetFileDescriptor;
import android.media.MediaPlayer;
import android.media.MediaPlayer.OnCompletionListener;

import com.badlogic.androidgames.framework.Music;

public class AndroidMusic implements Music, OnCompletionListener {
    MediaPlayer mediaPlayer;
    boolean isPrepared = false; package com.badlogic.androidgames.framework.impl;

import java.io.IOException;

import android.content.res.AssetFileDescriptor;
import android.media.MediaPlayer;
import android.media.MediaPlayer.OnCompletionListener;

import com.badlogic.androidgames.framework.Music;

public class AndroidMusic implements Music, OnCompletionListener {
    MediaPlayer mediaPlayer;
    boolean isPrepared = false;
```

The `AndroidMusic` class stores a `MediaPlayer` instance along with a boolean called `isPrepared`. Remember, we can only call `MediaPlayer.start()/stop()/pause()` when the `MediaPlayer` is prepared. This member helps us keep track of the `MediaPlayer`'s state.

The `AndroidMusic` class implements not only the `Music` interface, but also the `OnCompletionListener` interface. In Chapter 3 we briefly defined this interface as a means to get informed about when a `MediaPlayer` has stopped playing back a music file. If this happens, then the `MediaPlayer` needs to be prepared again before we can invoke any of the other methods on it. The method `OnCompletionListener.onCompletion()` might be called in a separate thread, and since we set the `isPrepared` member in this method, we have to make sure that it is safe from concurrent modifications.

```
    public AndroidMusic(AssetFileDescriptor assetDescriptor) {
        mediaPlayer = new MediaPlayer();
        try {
            mediaPlayer.setDataSource(assetDescriptor.getFileDescriptor(),
                    assetDescriptor.getStartOffset(),
                    assetDescriptor.getLength());
            mediaPlayer.prepare();
            isPrepared = true;
            mediaPlayer.setOnCompletionListener(this);
        } catch (Exception e) {
            throw new RuntimeException("Couldn't load music");
        }
    }
```

In the constructor we create and prepare the `MediaPlayer` from the `AssetFileDescriptor` that gets passed in, and we set the `isPrepared` flag, along with registering the `AndroidMusic` instance as an `OnCompletionListener` with the `MediaPlayer`. If anything goes wrong, we again throw an unchecked `RuntimeException`.

```
    @Override
    public void dispose() {
        if (mediaPlayer.isPlaying())
            mediaPlayer.stop();
        mediaPlayer.release();
    }
```

The dispose() method first checks if the MediaPlayer is still playing, and if so, stops it. Otherwise the call to MediaPlayer.release() would throw a runtime exception.

```
    @Override
    public boolean isLooping() {
        return mediaPlayer.isLooping();
    }

    @Override
    public boolean isPlaying() {
        return mediaPlayer.isPlaying();
    }

    @Override
    public boolean isStopped() {
        return !isPrepared;
    }
```

The methods isLooping(), isPlaying(), and isStopped() are straightforward. The first two use methods provided by the MediaPlayer; the last one uses the isPrepared flag, which indicates if the MediaPlayer is stopped—something MediaPlayer.isPlaying() does not necessarily tell us, as it returns false in case the MediaPlayer is paused but not stopped.

```
    @Override
    public void play() {
        if (mediaPlayer.isPlaying())
            return;

        try {
            synchronized (this) {
                if (!isPrepared)
                    mediaPlayer.prepare();
                mediaPlayer.start();
            }
        } catch (IllegalStateException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

The play() method is a little involved. If we are already playing, we simply return from the function. Next we have a mighty try...catch block within which we first check if the MediaPlayer is already prepared based on our flag, and prepare it if needed. If all goes well, we call the MediaPlayer.start() method, which will start the playback. All this is done in a synchronized block, as we use the isPrepared flag, which might get set on a separate thread due to our implementing the OnCompletionListener interface. In case something goes wrong, we again throw an unchecked RuntimeException.

```
@Override
    public void setLooping(boolean isLooping) {
        mediaPlayer.setLooping(isLooping);
    }

    @Override
    public void setVolume(float volume) {
        mediaPlayer.setVolume(volume, volume);
    }
```

The setLooping() and setVolume() methods can be called in any state of the MediaPlayer, and just delegate to the respective MediaPlayer methods.

```
@Override
    public void stop() {
        mediaPlayer.stop();
        synchronized (this) {
            isPrepared = false;
        }
    }
```

The stop() method stops the MediaPlayer and sets the isPrepared flag in a synchronized block again.

```
@Override
    public void onCompletion(MediaPlayer player) {
        synchronized (this) {
            isPrepared = false;
        }
    }
}
```

Finally there's the OnCompletionListener.onCompletion() method that the AndroidMusic class implements. All it does is set the isPrepared flag in a synchronized block so the other methods don't start throwing exceptions out of the blue.

Next we'll move on to our input-related classes.

AndroidInput and AccelerometerHandler

The Input interface we designed in Chapter 3 grants us access to the accelerometer, the touchscreen and the keyboard in polling and event modes via a couple of convenient methods. Putting all the code for an implementation of that interface into a single file is a bit nasty, though, so we will outsource all the input event handling into handler classes. The Input implementation will then use those handlers to pretend that it is actually performing all the work.

AccelerometerHandler: Which Side Is Up?

Let's start with the easiest of all handlers: the `AccelerometerHandler`. Listing 5–5 shows you its code.

Listing 5–5. *AccelerometerHandler.java; Performing All the Accelerometer Handling*

```
package com.badlogic.androidgames.framework.impl;

import android.content.Context;
import android.hardware.Sensor;
import android.hardware.SensorEvent;
import android.hardware.SensorEventListener;
import android.hardware.SensorManager;

public class AccelerometerHandler implements SensorEventListener {
    float accelX;
    float accelY;
    float accelZ;

    public AccelerometerHandler(Context context) {
        SensorManager manager = (SensorManager) context
                .getSystemService(Context.SENSOR_SERVICE);
        if (manager.getSensorList(Sensor.TYPE_ACCELEROMETER).size() != 0) {
            Sensor accelerometer = manager.getSensorList(
                    Sensor.TYPE_ACCELEROMETER).get(0);
            manager.registerListener(this, accelerometer,
                    SensorManager.SENSOR_DELAY_GAME);
        }
    }

    @Override
    public void onAccuracyChanged(Sensor sensor, int accuracy) {
        // nothing to do here
    }

    @Override
    public void onSensorChanged(SensorEvent event) {
        accelX = event.values[0];
        accelY = event.values[1];
        accelZ = event.values[2];
    }

    public float getAccelX() {
        return accelX;
    }

    public float getAccelY() {
        return accelY;
    }

    public float getAccelZ() {
        return accelZ;
    }
}
```

Unsurprisingly, the class implements the SensorEventListener interface, which we used in Chapter 4. The class stores three members, holding the acceleration on each of the three accelerometer axes.

The constructor takes a Context, from which it gets a SensorManager instance to set up the event listening. The rest of the code is equivalent to what we did in the last chapter. Note that if no accelerometer is installed, the handler will happily return zero acceleration on all axes throughout its life. We thus don't need any extra error-checking or exception-throwing code.

The next two methods, onAccuracyChanged() and onSensorChanged(), should also be familiar. In the first we don't do anything, as there's nothing much to report. In the second one we fetch the accelerometer values from the provided SensorEvent and store them in the handler's members.

The final three methods simply return the current acceleration for each axis.

Note that we do not need to perform any synchronization here, even though the onSensorChanged() method might be called in a different thread. The Java memory model guarantees that writes and reads to and from primitive types such as boolean, int, or byte are atomic. In this case it's OK to rely on this fact, as we don't do anything more complex than assigning a new value. We'd need to have proper synchronization if this were not the case (e.g., if we did something with the member variables in the onSensorChanged() method.

The Pool Class: Because Reuse is Good for You!

What's the worst thing that can happen to us as Android developers? World-stopping garbage collection! If you look at the Input interface definition in Chapter 3, you'll find the methods getTouchEvents() and getKeyEvents(). These return lists of TouchEvents and KeyEvents. In our keyboard and touch event handlers, we'll constantly create instances of these two classes and store them in lists internal to the handlers. The Android input system fires a lot of those events when a key is pressed or a finger is touching the screen, so we'd constantly create new instances that will get collected by the garbage collector in short intervals. In order to avoid this, we will implement a concept known as *instance pooling*. Instead of creating new instances of a class frequently, we'll simply reuse previously created instances. The Pool class is a convenient way to implement that behavior. Let's have a look at its code in Listing 5–6.

Listing 5–6. ***Pool.java; Playing Well with the Garbage Collector***

```
package com.badlogic.androidgames.framework;

import java.util.ArrayList;
import java.util.List;

public class Pool<T> {
```

Here come generics: the first thing to recognize is that this class is a generically typed class, much like collection classes, such as ArrayList or HashMap. Generics allow us to store any type of object in our Pool without having to cast like mad. So what does the Pool class do?

```
    public interface PoolObjectFactory<T> {
        public T createObject();
    }
```

The first thing that's defined is an interface called PoolObjectFactory, which is again generic. It has a single method, createObject(), which will return a shiny new object that has the generic type of the Pool/PoolObjectFactory instance.

```
    private final List<T> freeObjects;
    private final PoolObjectFactory<T> factory;
    private final int maxSize;
```

The Pool class has three members: an ArrayList to store pooled objects, a PoolObjectFactory that is used to generate new instances of the type the class holds, and a member that stores the maximum number of objects the Pool can hold. The last bit is needed so that our Pool does not grow indefinitely; otherwise we might run into an out-of-memory exception.

```
    public Pool(PoolObjectFactory<T> factory, int maxSize) {
        this.factory = factory;
        this.maxSize = maxSize;
        this.freeObjects = new ArrayList<T>(maxSize);
    }
```

The constructor of the Pool class takes a PoolObjectFactory and the maximum number of objects it should store. We store both parameters in the respective members and instantiate a new ArrayList with the capacity set to the maximum number of objects.

```
    public T newObject() {
        T object = null;

        if (freeObjects.size() == 0)
            object = factory.createObject();
        else
            object = freeObjects.remove(freeObjects.size() - 1);

        return object;
    }
```

The newObject() method is responsible for either handing us a brand-new instance of the type that the Pool holds via the PoolObjectFactory.newObject() method, or returning a pooled instance in case there's one in the freeObjects ArrayList. If we use this method, we'll get recycled objects as long as the Pool has some stored in the freeObjects list. Otherwise the method will create a new one via the factory.

```
    public void free(T object) {
        if (freeObjects.size() < maxSize)
            freeObjects.add(object);
    }
}
```

The free() method lets us reinsert objects we no longer use. All it does is insert the object into the freeObjects list if it is not filled to capacity yet. If the list is full, the object is not added, and is likely to be consumed by the garbage collector the next time it executes.

So how can we use that class? Let's look at some pseudocode usage of the Pool class in conjunction with touch events:

```
PoolObjectFactory<TouchEvent> factory = new PoolObjectFactory<TouchEvent>() {
    @Override
    public TouchEvent createObject() {
        return new TouchEvent();
    }
};
Pool<TouchEvent> touchEventPool = new Pool<TouchEvent>(factory, 50);
TouchEvent touchEvent = touchEventPool.newObject();
… do something here …
touchEventPool.free(touchEvent);
```

We first define a PoolObjectFactory that creates TouchEvent instances. Next we instantiate the Pool, telling it to use our factory and that it should maximally store 50 TouchEvents. When we want a new TouchEvent from the Pool, we call the Pool's newObject() method. Initially the Pool is empty, so it will ask the factory to create a brand-new TouchEvent instance. When we no longer need the TouchEvent, we can reinsert it into the Pool by calling the Pool's free() method. The next time we call the newObject() method, we will get the same TouchEvent instance again, recycling it so the garbage collector doesn't get mad at us. That class will come in handy in a couple of places. Just note that you have to take care if you reuse objects: it's easy to not fully reinitialize them when they're fetched from the Pool.

KeyboardHandler: Up, Up, Down, Down, Left, Right . . .

The KeyboardHandler has to fulfill a couple of tasks. First it must hook up with the View from which keyboard events are to be received. Next it must store the current state of each key for polling. It must also keep a list of KeyEvent instances, which we designed in Chapter 3 for event-based input handling. Finally it must properly synchronize all this, as it will receive events on the UI thread while being polled from our main game loop, which is executed on a different thread. Quite a lot of work. As a little refresher, let me show you the KeyEvent class again, which we defined in Chapter 3 as part of the Input interface:

```
public static class KeyEvent {
    public static final int KEY_DOWN = 0;
    public static final int KEY_UP = 1;

    public int type;
    public int keyCode;
    public char keyChar;
}
```

It simply defines two constants encoding the key event type along with three members, holding the type, key code, and Unicode character of the event. With this we can implement our handler.

Listing 5–7 shows the implementation of the handler with the Android APIs discussed earlier and our new Pool class.

Listing 5–7. *KeyboardHandler.java: Handling Keys Since 2010*

```
package com.badlogic.androidgames.framework.impl;

import java.util.ArrayList;
import java.util.List;

import android.view.View;
import android.view.View.OnKeyListener;

import com.badlogic.androidgames.framework.Input.KeyEvent;
import com.badlogic.androidgames.framework.Pool;
import com.badlogic.androidgames.framework.Pool.PoolObjectFactory;

public class KeyboardHandler implements OnKeyListener {
    boolean[] pressedKeys = new boolean[128];
    Pool<KeyEvent> keyEventPool;
    List<KeyEvent> keyEventsBuffer = new ArrayList<KeyEvent>();
    List<KeyEvent> keyEvents = new ArrayList<KeyEvent>();
```

The KeyboardHandler class implements the OnKeyListener interface so that it can receive key events from a View. Next up are the members.

The first member is an array holding 128 booleans. We'll store the current state (pressed or not) of each key in this array. It is indexed by the key code of a key. Luckily for us, the android.view.KeyEvent.KEYCODE_XXX constants (which encode the key codes) are all in the range between 0 and 127, so we can store them in this garbage collector–friendly form. Note that by an unlucky accident our KeyEvent class shares its name with the Android KeyEvent class, instances of which get passed to our OnKeyEventListener.onKeyEvent() method. This slight confusion is limited to this handler code only. As there's hardly a better name for a key event than KeyEvent, we chose to live with this short-lived confusion.

The next member is a Pool that holds instances of our KeyEvent class. We don't want to make the garbage collector angry, so we recycle all the KeyEvent objects we create.

The third member stores the KeyEvents that have not yet been consumed by our game. Each time we get a new key event on the UI thread we'll add it to this list.

The last member stores the KeyEvents we'll return upon a call to KeyboardHandler.getKeyEvents(). We'll see why we have to double-buffer the key events in a minute.

```
    public KeyboardHandler(View view) {
        PoolObjectFactory<KeyEvent> factory = new PoolObjectFactory<KeyEvent>() {
            @Override
            public KeyEvent createObject() {
                return new KeyEvent();
            }
        };
        keyEventPool = new Pool<KeyEvent>(factory, 100);
        view.setOnKeyListener(this);
        view.setFocusableInTouchMode(true);
        view.requestFocus();
    }
```

The constructor has a single parameter consisting of the View we want to receive key events from. We create the Pool instance with a proper PoolObjectFactory, register the handler as an OnKeyListener with the View, and finally make sure that the View will receive key events by making it the focused View.

```
    @Override
    public boolean onKey(View v, int keyCode, android.view.KeyEvent event) {
        if (event.getAction() == android.view.KeyEvent.ACTION_MULTIPLE)
            return false;

        synchronized (this) {
            KeyEvent keyEvent = keyEventPool.newObject();
            keyEvent.keyCode = keyCode;
            keyEvent.keyChar = (char) event.getUnicodeChar();
            if (event.getAction() == android.view.KeyEvent.ACTION_DOWN) {
                keyEvent.type = KeyEvent.KEY_DOWN;
                if(keyCode > 0 && keyCode < 127)
                    pressedKeys[keyCode] = true;
            }
            if (event.getAction() == android.view.KeyEvent.ACTION_UP) {
                keyEvent.type = KeyEvent.KEY_UP;
                if(keyCode > 0 && keyCode < 127)
                    pressedKeys[keyCode] = false;
            }
            keyEventsBuffer.add(keyEvent);
        }
        return false;
    }
```

Next up is our implementation of the OnKeyListener.onKey() interface method, which gets called each time the View receives a new key event. We start by ignoring any (Android) key events that encode a KeyEvent.ACTION_MULTIPLE event. These are not relevant in our context. We follow that up with a tasty synchronized block. Remember that the events are received on the UI thread and read on the main loop thread, so we have to make sure none of our members are accessed in parallel.

Within the synchronized block we first fetch a KeyEvent instance (of our KeyEvent implementation) from the Pool. This will either get us a recycled instance or a brand-new one, depending on the state of the Pool. Next we set the KeyEvent's keyCode and keyChar members based on the contents of the Android KeyEvent that got passed to the method. We then decode the type of the Android KeyEvent and set the type of our KeyEvent as well as the element in the pressedKey array accordingly. Finally we add our KeyEvent to the keyEventBuffer list we defined earlier.

```
    public boolean isKeyPressed(int keyCode) {
        if (keyCode < 0 || keyCode > 127)
            return false;
        return pressedKeys[keyCode];
    }
```

Next we have the `isKeyPressed()` method, which basically implements the semantics of `Input.isKeyPressed()`. We pass in an integer specifying the key code (one of the Android `KeyEvent.KEYCODE_XXX` constants) and return whether that key is pressed or not. We do so by looking up the state of the key in the `pressedKey` array after some range checking. Remember that we set the elements of this array in the previous method, which gets called on the UI thread. As we are again working with primitive types, there's no need for synchronization.

```
    public List<KeyEvent> getKeyEvents() {
        synchronized (this) {
            int len = keyEvents.size();
            for (int i = 0; i < len; i++)
                keyEventPool.free(keyEvents.get(i));
            keyEvents.clear();
            keyEvents.addAll(keyEventsBuffer);
            keyEventsBuffer.clear();
            return keyEvents;
        }
    }
}
```

The last method of our handler is called `getKeyEvents()`, and implements the semantics of the `Input.getKeyEvents()` method. We start off with a juicy synchronized block again, remembering that this method will be called from a different thread.

Next we do something very mysterious. We loop through the `keyEvents` array and insert all the `KeyEvents` stored in it into our `Pool`. Remember that we fetch instances from the `Pool` in the `onKey()` method on the UI thread. Here we reinsert them into the `Pool`. But isn't the `keyEvents` list empty? Yes, but only the first time we invoke that method. To understand why that is, you have to grasp the rest of the method first.

After our mysterious `Pool` insertion loop, we clear the `keyEvents` list and fill it with the events in our `keyEventsBuffer` list. Finally we clear the `keyEventsBuffer` list and return the newly filled `keyEvents` list to the caller. What is happening here?

Let me illustrate it by giving you a simple example. We'll examine what happens to the `keyEvents` and `keyEventsBuffer` lists, as well as our `Pool` each time a new event arrives on the UI thread or the game is fetching the events in the main thread:

```
UI thread: onKey() ->
           keyEvents = { }, keyEventsBuffer = {KeyEvent1}, pool = { }
Main thread: getKeyEvents() ->
           keyEvents = {KeyEvent1}, keyEventsBuffer = { }, pool { }
UI thread: onKey() ->
           keyEvents = {KeyEvent1}, keyEventsBuffer = {KeyEvent2}, pool { }
Main thread: getKeyEvents() ->
           keyEvents = {KeyEvent2}, keyEventsBuffer = { }, pool = {KeyEvent1}
UI thread: onKey() ->
```

```
keyEvents = {KeyEvent2}, keyEventsBuffer = {KeyEvent1}, pool = { }
```

1. First we get a new event in the UI thread. There's nothing in the `Pool` yet, so a new `KeyEvent` instance (`KeyEvent1`) is created and inserted into the `keyEventsBuffer` list.
2. Next we call `getKeyEvents()` on the main thread. It takes `KeyEvent1` from the `keyEventsBuffer` list and puts it into the `keyEvents` list it returns to the caller.
3. We get another event on the UI thread. We still have nothing in the `Pool`, so a new `KeyEvent` instance (`KeyEvent2`) is created and inserted into the `keyEventsBuffer` list.
4. The main thread calls `getKeyEvents()` again. Now something interesting happens. Upon entry into the method, the `keyEvents` list still holds `KeyEvent1`. The mysterious insertion loop will place that event into our `Pool`. It then clears the `keyEvents` list and inserts any `KeyEvent` into the `keyEventsBuffer`, in this case `KeyEvent2`. We just recycled a key event.
5. Finally another key event arrives on the UI thread. This time we have a free `KeyEvent` in our `Pool`, which we'll happily reuse. Look mom, no garbage collection!

This mechanism comes with one caveat, though: we have to call `KeyboardHandler.getKeyEvents()` frequently or else the `keyEvents` list will fill up quickly, and no objects will be returned to the `Pool`. As long as we remember this, all is well.

Touch Handlers

And now fragmentation hits us. In the last chapter we talked a little about the fact that multitouch is supported on Android versions greater than 1.6 only. All the nice constants we used in our multitouch code (e.g., `MotionEvent.ACTION_POINTER_ID_MASK`) are not available to us on Android 1.5 or 1.6. While we can use them in our code just fine if we set the build target of our project to an Android version that has this API, the application will crash on any device running Android 1.5 or 1.6. We want our games to run on all currently available Android versions, so how do we solve this problem?

We employ a simple trick. We write two handlers, one using the single-touch API of Android 1.5 and another using the multitouch API of Android 2.0 and above. As long as we don't execute the code of the multitouch handler on a device with a lower Android version than 2.0, we are safe. The code won't get loaded by the VM, and it won't throw exceptions like crazy. All we need to do is to find out which Android version the device is running and instantiate the proper handler. You'll see how that works when we discuss the `AndroidInput` class. For now let's concentrate on the two handlers.

The TouchHandler Interface

In order to be able to use our two handler classes interchangeably, we need to define a common interface. Listing 5–8 shows this interface, called `TouchHandler`.

Listing 5–8. ***TouchHandler.java, to Be Implemented for Android 1.5 and 1.6.***

```
package com.badlogic.androidgames.framework.impl;

import java.util.List;

import android.view.View.OnTouchListener;

import com.badlogic.androidgames.framework.Input.TouchEvent;

public interface TouchHandler extends OnTouchListener {
    public boolean isTouchDown(int pointer);

    public int getTouchX(int pointer);

    public int getTouchY(int pointer);

    public List<TouchEvent> getTouchEvents();
}
```

All `TouchHandlers` must also implement the `OnTouchListener` interface, which we use to register the handler with a `View`. The methods of the interface correspond to the respective methods in the `Input` interface defined in Chapter 3. The first three are for polling the state of a specific pointer, and the last one is for getting `TouchEvents` so we can do event-based input handling. Note that the polling methods take a pointer ID.

The SingleTouchHandler Class

In the case of our single-touch handler, we'll ignore any IDs other than zero. As a refresher, let's recall the `TouchEvent` class defined in Chapter 3 as part of the `Input` interface:

```
public static class TouchEvent {
    public static final int TOUCH_DOWN = 0;
    public static final int TOUCH_UP = 1;
    public static final int TOUCH_DRAGGED = 2;

    public int type;
    public int x, y;
    public int pointer;
}
```

Like the `KeyEvent` class, it defines a couple of constants encoding the touch event's type, along with the x- and y-coordinates in the coordinate system of the `View` and the pointer ID.

Listing 5–9 shows the implementation of the `TouchHandler` interface for Android 1.5 and 1.6.

Listing 5–9. *SingleTouchHandler.java; Good with Single Touch, Not So Good with Multitouch*

```
package com.badlogic.androidgames.framework.impl;

import java.util.ArrayList;
import java.util.List;

import android.view.MotionEvent;
import android.view.View;

import com.badlogic.androidgames.framework.Pool;
import com.badlogic.androidgames.framework.Input.TouchEvent;
import com.badlogic.androidgames.framework.Pool.PoolObjectFactory;

public class SingleTouchHandler implements TouchHandler {
    boolean isTouched;
    int touchX;
    int touchY;
    Pool<TouchEvent> touchEventPool;
    List<TouchEvent> touchEvents = new ArrayList<TouchEvent>();
    List<TouchEvent> touchEventsBuffer = new ArrayList<TouchEvent>();
    float scaleX;
    float scaleY;
```

We start off by letting the class implement the `TouchHandler` interface, which also means that we have to implement the `OnTouchListener` interface. Next are a couple of members that should look familiar. We have three members storing the current state of the touchscreen for one finger, followed by a `Pool` and two lists holding `TouchEvents`. This is exactly the same thing we had in the `KeyboardHandler`. We also have two members, `scaleX` and `scaleY`. We'll talk about those in a minute. We'll use these to cope with different screen resolutions.

NOTE: Of course, we could have made that more elegant by letting the `KeyboardHandler` and `SingleTouchHandler` derive from a base class that handles all this pooling and synchronization stuff. It would have complicated the explanation even more, though, so instead we'll just write a few more lines of code.

```
public SingleTouchHandler(View view, float scaleX, float scaleY) {
        PoolObjectFactory<TouchEvent> factory = new PoolObjectFactory<TouchEvent>() {
            @Override
            public TouchEvent createObject() {
                return new TouchEvent();
            }
        };
        touchEventPool = new Pool<TouchEvent>(factory, 100);
        view.setOnTouchListener(this);

        this.scaleX = scaleX;
        this.scaleY = scaleY;
    }
```

In the constructor we register the handler as an `OnTouchListener` and set up the `Pool` we use to recycle `TouchEvents`. We also store the `scaleX` and `scaleY` parameters that are passed to the constructor (and ignore them for now).

```
@Override
    public boolean onTouch(View v, MotionEvent event) {
        synchronized(this) {
            TouchEvent touchEvent = touchEventPool.newObject();
            switch (event.getAction()) {
            case MotionEvent.ACTION_DOWN:
                touchEvent.type = TouchEvent.TOUCH_DOWN;
                isTouched = true;
                break;
            case MotionEvent.ACTION_MOVE:
                touchEvent.type = TouchEvent.TOUCH_DRAGGED;
                isTouched = true;
                break;
            case MotionEvent.ACTION_CANCEL:
            case MotionEvent.ACTION_UP:
                touchEvent.type = TouchEvent.TOUCH_UP;
                isTouched = false;
                break;
            }

            touchEvent.x = touchX = (int)(event.getX() * scaleX);
            touchEvent.y = touchY = (int)(event.getY() * scaleY);
            touchEventsBuffer.add(touchEvent);

            return true;
        }
    }
```

The `onTouch()` method does the same thing as the `onKey()` method of our `KeyboardHandler`, the only difference being that we now handle `TouchEvents`, not `KeyEvents`. All the synchronization, pooling, and `MotionEvent` handling are already known to us. The only interesting thing is that we actually multiply the reported x- and y-coordinates of a touch event by `scaleX` and `scaleY`. Remember this, as we'll take a look at it again later on.

```
@Override
    public boolean isTouchDown(int pointer) {
        synchronized(this) {
            if(pointer == 0)
                return isTouched;
            else
                return false;
        }
    }

    @Override
    public int getTouchX(int pointer) {
        synchronized(this) {
            return touchX;
        }
    }
    @Override
    public int getTouchY(int pointer) {
        synchronized(this) {
            return touchY;
        }
    }
```

The methods `isTouchDown()`, `getTouchX()`, and `getTouchY()` allow us to poll the touchscreen state based on the members that we set in the `onTouch()` method. The only noticeable thing about them is that they'll only return useful data for a pointer ID with a value zero, as we only support single-touch screens with this class.

```
@Override
    public List<TouchEvent> getTouchEvents() {
        synchronized(this) {
            int len = touchEvents.size();
            for( int i = 0; i < len; i++ )
                touchEventPool.free(touchEvents.get(i));
            touchEvents.clear();
            touchEvents.addAll(touchEventsBuffer);
            touchEventsBuffer.clear();
            return touchEvents;
        }
    }
}
```

The final method, `SingleTouchHandler.getTouchEvents()`, should be familiar to you, and works similarly to the `KeyboardHandler.getKeyEvents()` methods. Remember that we need to call this method frequently so that the `touchEvents` list doesn't get filled up.

The MultiTouchHandler

For multitouch handling, we have a class called `MultiTouchHandler`, as shown in Listing 5–10.

Listing 5–10. *MultiTouchHandler.java (More of the Same)*

```
package com.badlogic.androidgames.framework.impl;

import java.util.ArrayList;
import java.util.List;

import android.view.MotionEvent;
import android.view.View;

import com.badlogic.androidgames.framework.Input.TouchEvent;
import com.badlogic.androidgames.framework.Pool;
import com.badlogic.androidgames.framework.Pool.PoolObjectFactory;

public class MultiTouchHandler implements TouchHandler {
    boolean[] isTouched = new boolean[20];
    int[] touchX = new int[20];
    int[] touchY = new int[20];
    Pool<TouchEvent> touchEventPool;
    List<TouchEvent> touchEvents = new ArrayList<TouchEvent>();
    List<TouchEvent> touchEventsBuffer = new ArrayList<TouchEvent>();
    float scaleX;
    float scaleY;
```

We again let the class implement the `TouchHandler` interface and have a couple of members to store the current state and events. Instead of storing the state for a single pointer, we simply store the state of 20 pointers. We also have those mysterious `scaleX` and `scaleY` members again.

```
public MultiTouchHandler(View view, float scaleX, float scaleY) {
        PoolObjectFactory<TouchEvent> factory = new PoolObjectFactory<TouchEvent>() {
            @Override
            public TouchEvent createObject() {
                return new TouchEvent();
            }
        };
        touchEventPool = new Pool<TouchEvent>(factory, 100);
        view.setOnTouchListener(this);

        this.scaleX = scaleX;
        this.scaleY = scaleY;
    }
```

The constructor is exactly the same as the constructor of the `SingleTouchHandler`: we create a `Pool` for `TouchEvent` instances register the handler as an `OnTouchListener`, and store the scaling values.

```
@Override
    public boolean onTouch(View v, MotionEvent event) {
        synchronized (this) {
            int action = event.getAction() & MotionEvent.ACTION_MASK;
            int pointerIndex = (event.getAction() & MotionEvent.ACTION_POINTER_ID_MASK)
>> MotionEvent.ACTION_POINTER_ID_SHIFT;
            int pointerId = event.getPointerId(pointerIndex);
            TouchEvent touchEvent;

            switch (action) {
            case MotionEvent.ACTION_DOWN:
            case MotionEvent.ACTION_POINTER_DOWN:
                touchEvent = touchEventPool.newObject();
                touchEvent.type = TouchEvent.TOUCH_DOWN;
                touchEvent.pointer = pointerId;
                touchEvent.x = touchX[pointerId] = (int) (event
                        .getX(pointerIndex) * scaleX);
                touchEvent.y = touchY[pointerId] = (int) (event
                        .getY(pointerIndex) * scaleY);
                isTouched[pointerId] = true;
                touchEventsBuffer.add(touchEvent);
                break;

            case MotionEvent.ACTION_UP:
            case MotionEvent.ACTION_POINTER_UP:
            case MotionEvent.ACTION_CANCEL:
                touchEvent = touchEventPool.newObject();
                touchEvent.type = TouchEvent.TOUCH_UP;
                touchEvent.pointer = pointerId;
                touchEvent.x = touchX[pointerId] = (int) (event
                        .getX(pointerIndex) * scaleX);
                touchEvent.y = touchY[pointerId] = (int) (event
                        .getY(pointerIndex) * scaleY);
                isTouched[pointerId] = false;
                touchEventsBuffer.add(touchEvent);
                break;

            case MotionEvent.ACTION_MOVE:
                int pointerCount = event.getPointerCount();
                for (int i = 0; i < pointerCount; i++) {
                    pointerIndex = i;
                    pointerId = event.getPointerId(pointerIndex);

                    touchEvent = touchEventPool.newObject();
                    touchEvent.type = TouchEvent.TOUCH_DRAGGED;
                    touchEvent.pointer = pointerId;
                    touchEvent.x = touchX[pointerId] = (int) (event
                            .getX(pointerIndex) * scaleX);
                    touchEvent.y = touchY[pointerId] = (int) (event
                            .getY(pointerIndex) * scaleY);
                    touchEventsBuffer.add(touchEvent);
                }
                break;
            }

            return true;
        }
    }
```

The `onTouch()` method looks as intimidating as in our test example in the last chapter. All we do is marry that test code with our event pooling and synchronization here (things we've already talked about in detail). The only real difference to the `SingleTouchHandler.onTouch()` method is that we handle multiple pointers and set the `TouchEvent.pointer` member accordingly (instead of just to zero).

```
@Override
    public boolean isTouchDown(int pointer) {
        synchronized (this) {
            if (pointer < 0 || pointer >= 20)
                return false;
            else
                return isTouched[pointer];
        }
    }

    @Override
    public int getTouchX(int pointer) {
        synchronized (this) {
            if (pointer < 0 || pointer >= 20)
                return 0;
            else
                return touchX[pointer];
        }
    }

    @Override
    public int getTouchY(int pointer) {
        synchronized (this) {
            if (pointer < 0 || pointer >= 20)
                return 0;
            else
                return touchY[pointer];
        }
    }
```

The polling methods `isTouchDown()`, `getTouchX()`, and `getTouchY()` should look familiar as well. We perform some error checking and then fetch the corresponding pointer state from one of the member arrays that we fill in the `onTouch()` method.

```
@Override
    public List<TouchEvent> getTouchEvents() {
        synchronized (this) {
            int len = touchEvents.size();
            for (int i = 0; i < len; i++)
                touchEventPool.free(touchEvents.get(i));
            touchEvents.clear();
            touchEvents.addAll(touchEventsBuffer);
            touchEventsBuffer.clear();
            return touchEvents;
        }
    }
}
```

The final method, `getTouchEvents()`, is again exactly the same as the corresponding method of `SingleTouchHandler.getTouchEvents()`.

Equipped with all those handlers, we can now implement the `Input` interface.

AndroidInput: The Great Coordinator

The `Input` implementation of our game framework ties together all the handlers we just developed. Any method calls will be delegated to the corresponding handler. The only interesting part of this implementation is where we choose which `TouchHandler` implementation we use based on the Android version the device is running. Listing 5–11 shows you the implementation, called `AndroidInput`.

Listing 5–11. ***AndroidInput.java; Handling the Handlers with Style***

```
package com.badlogic.androidgames.framework.impl;

import java.util.List;

import android.content.Context;
import android.os.Build.VERSION;
import android.view.View;

import com.badlogic.androidgames.framework.Input;
```

```
public class AndroidInput implements Input {
    AccelerometerHandler accelHandler;
    KeyboardHandler keyHandler;
    TouchHandler touchHandler;
```

We start off by letting the class implement the Input interface we defined in Chapter 3. Next we find three members: an AccelerometerHandler, a KeyboardHandler, and a TouchHandler.

```
public AndroidInput(Context context, View view, float scaleX, float scaleY) {
        accelHandler = new AccelerometerHandler(context);
        keyHandler = new KeyboardHandler(view);
        if(Integer.parseInt(VERSION.SDK) < 5)
            touchHandler = new SingleTouchHandler(view, scaleX, scaleY);
        else
            touchHandler = new MultiTouchHandler(view, scaleX, scaleY);
    }
```

These members get initialized in the constructor, which takes a Context, a View, and those scaleX and scaleY parameters that we can happily ignore again. The AccelerometerHandler gets instantiated via the Context parameter, and the KeyboardHandler needs the View that gets passed in.

To decide which TouchHandler to use, we simply check the Android version the application runs on. This can be done via the VERSION.SDK string, a constant provided by the Android API. Why it is a string is unclear, as it directly encodes the SDK version numbers we use in our manifest file. We therefore need to make it an integer to do some comparisons. The first Android version to support the multitouch API was version 2.0, which corresponds to SDK version 5. If the current device runs an Android version below that, we instantiate the SingleTouchHandler; otherwise we use the MultiTouchHandler. And that's all the fragmentation we have to care about at an API level. When we start doing OpenGL rendering, we'll hit a few more of these fragmentation issues—but don't worry, they can be as easily resolved as the touch API problems.

```
@Override
    public boolean isKeyPressed(int keyCode) {
        return keyHandler.isKeyPressed(keyCode);
    }

    @Override
    public boolean isTouchDown(int pointer) {
        return touchHandler.isTouchDown(pointer);
    }

    @Override
    public int getTouchX(int pointer) {
        return touchHandler.getTouchX(pointer);
    }

    @Override
    public int getTouchY(int pointer) {
        return touchHandler.getTouchY(pointer);
    }

    @Override
    public float getAccelX() {
        return accelHandler.getAccelX();
    }

    @Override
    public float getAccelY() {
        return accelHandler.getAccelY();
    }

    @Override
    public float getAccelZ() {
        return accelHandler.getAccelZ();
    }

    @Override
    public List<TouchEvent> getTouchEvents() {
        return touchHandler.getTouchEvents();
    }

    @Override
    public List<KeyEvent> getKeyEvents() {
        return keyHandler.getKeyEvents();
    }
}
```

The rest of this class is more than self-explanatory. Each method call is delegated to the appropriate handler, which does the actual work. And with this, we have finished the input API of our little game framework. Next we'll move on to graphics.

AndroidGraphics and AndroidPixmap: Double Rainbow

It's time to get back to our most beloved topic: graphics programming. In Chapter 3 we defined two interfaces, `Graphics` and `Pixmap`; we are now going to implement them based on what you learned in Chapter 4. But there's one thing we have postponed until now: how to handle different screen sizes and resolutions.

Handling Different Screen Sizes and Resolutions

Android has supported different screen resolutions since version 1.6; it can handle resolutions ranging from 240! 320 pixels to a much beefier 480! 854 pixels on some new devices (in portrait mode; for landscape mode, just swap the values). In the last chapter we already saw the effect of these different screen resolutions and physical screen sizes: drawing with absolute coordinates and sizes given in pixels will produce unexpected results. Figure 5–1 shows you once more what happens when we render a 100! 100-pixel rectangle with the upper-left corner at (219,379) on 480! 800 and 320! 480 screens.

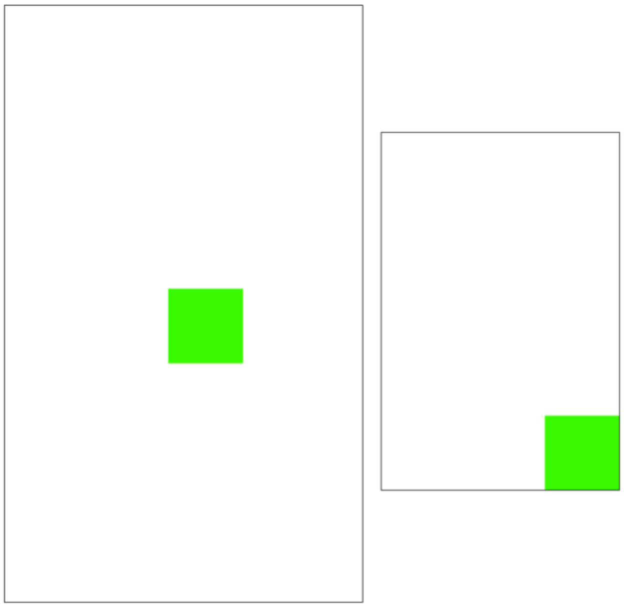

Figure 5–1. *A 100×100-pixel rectangle drawn at (219,379) on a 480×800 screen (left) and a 320×480 screen (right)*

This difference is bad for two reasons. First, we can't just draw our game assuming a fixed resolution. The second reason is subtler ,however. In Figure 5–1, I silently assumed that both screens have the same density (i.e., that each pixel has the same physical size on both devices), but this is hardly the case in reality.

Density

Density is usually specified in pixels per inch or pixels per centimeter (you'll sometimes also hear dots per inch, which is not technically exact). The Nexus One has a 480! 800-pixel screen with a physical size of 8! 4.8 centimeters. The HTC Hero has a 320! 480-pixel screen with a physical size of 6.5! 4.5 centimeters. That's 100 pixels per centimeter on both axes on the Nexus One, and roughly 71 pixels per centimeter on both axes on the Hero. We can calculate the pixels per centimeter easily like this:

```
pixels per centimeter (on x-axis) = width in pixels / width in centimeters
```

or this:

```
pixels per centimeter (on y-axis) = height in pixels / height in centimeters
```

Usually we only need to calculate this on a single axis, as the physical pixels are square (well, they're actually three pixels, but we'll just ignore that here).

How big would our 100! 100-pixel rectangle be in centimeters? On the Nexus One we'd have a 1! 1-centimeter rectangle, while on the Hero we'd have a 1.4! 1.4-centimeter rectangle. That's something we would need to account for if we had, for example, things like buttons that should be big enough for the average thumb on all screen sizes. However, while this example makes it look like this issue could present a huge problem, it usually doesn't. We just need to make sure that our buttons have a good size on high-density screens (e.g., the Nexus One). They will automatically be big enough on lower-density screens.

Aspect Ratio

There's also another problem we have to cope with, though: aspect ratio. The aspect ratio of a screen is the ratio between the width and height, either in pixels or centimeters. We can calculate that like this:

```
pixel aspect ratio = width in pixels / height in pixels
```

or this:

```
physical aspect ratio = width in centimeters / height in centimeters
```

When we use *width* and *height* here, we usually mean the width and height in landscape mode. The Nexus One has a pixel and physical aspect ratio of ~1.66. The Hero has a pixel and physical aspect ratio of 1.5. What does this mean? On the Nexus One we have more pixels available on the x-axis in landscape mode relative to the height than we have on the Hero. Figure 5–2 illustrates what this means with screenshots from Replica Island on both devices.

NOTE: In this book we'll use the metric system. I know that it might be a bit hard to get comfortable with if you are used to inches and pounds. However, as we will also do a little physics later on, which is usually defined in the metric system, it's best get used to it now. Just remember that 1 inch is roughly 2.54 centimeters.

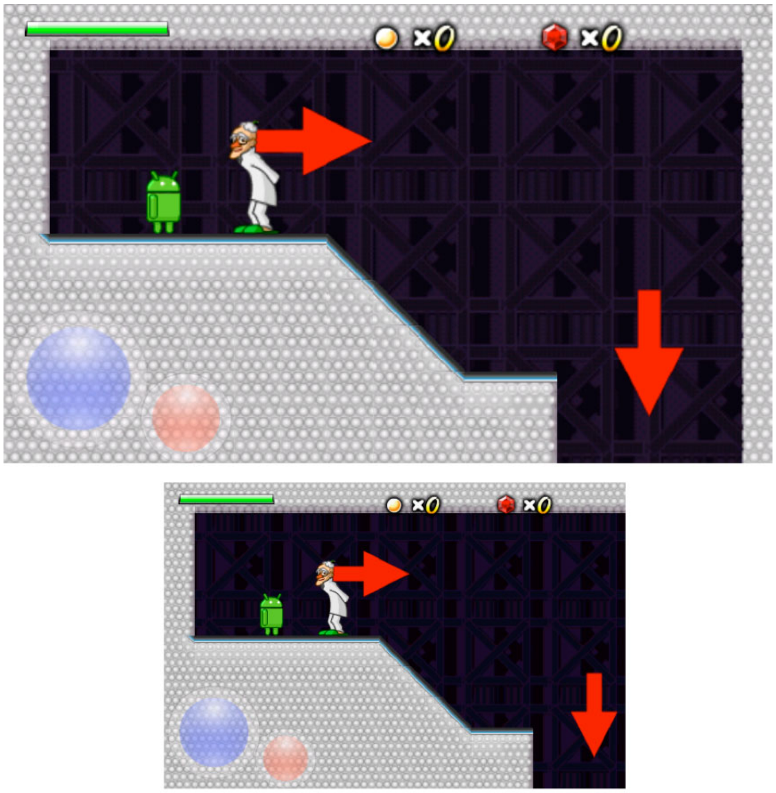

Figure 5–2. *Replica Island on the Nexus One (top) and the HTC Hero (bottom)*

The Nexus One displays a tiny bit more of the world on the x-axis. Everything stays the same on the y-axis though. Hmm, what did the creator of Replica Island do here?

Coping with Different Aspect Ratios

Replica Island performs a cheap but very useful magic trick in order to deal with the aspect ratio problem. The game was originally designed for everything to fit on a 480! 320-pixel screen, including all the sprites (e.g., the robot and the doctor), the tiles of the world, and the UI elements (e.g., the buttons at the bottom left and the status info at the top of the screen). When the game is rendered on a Hero, each pixel in the sprite bitmaps maps to exactly one pixel on the screen. On a Nexus One, everything is scaled up while rendering, so 1 pixel of a sprite actually takes up 1.5 pixels on the screen. In other words, a 32! 32-pixel sprite will be 48! 48 pixels big on the screen. This scaling factor can be easily calculated by

```
scaling factor (on x-axis) = screen width in pixels / target width in pixels
```

and

```
scaling factor (on y-axis) = screen height in pixels / target height in pixels
```

The target width and height equal the screen resolution that the graphical assets were designed for; in Replica Island, that's 480! 320 pixels. For the Nexus One, this means that we have a scaling factor of 1.66 on the x-axis and a scaling factor of 1.5 on the y-axis. But why are the scaling factors on the two axes different?

This is due to the two screen resolutions having different aspect ratios. If we simply stretch a 480! 320-pixel image to an 800! 480-pixel image, the original image will be stretched on the x-axis. For most games, this won't make too big of an impact, so we can simply draw our graphical assets for a specific target resolution and stretch them to the actual screen resolution on the fly while rendering (remember the `Bitmap.drawBitmap()` method).

For some games, however, you might want to get a little fancier. Figure 5–3 shows Replica Island simply scaled up from 480! 320 to 800! 480 pixels, and overlaid with a faint image of how it actually looks.

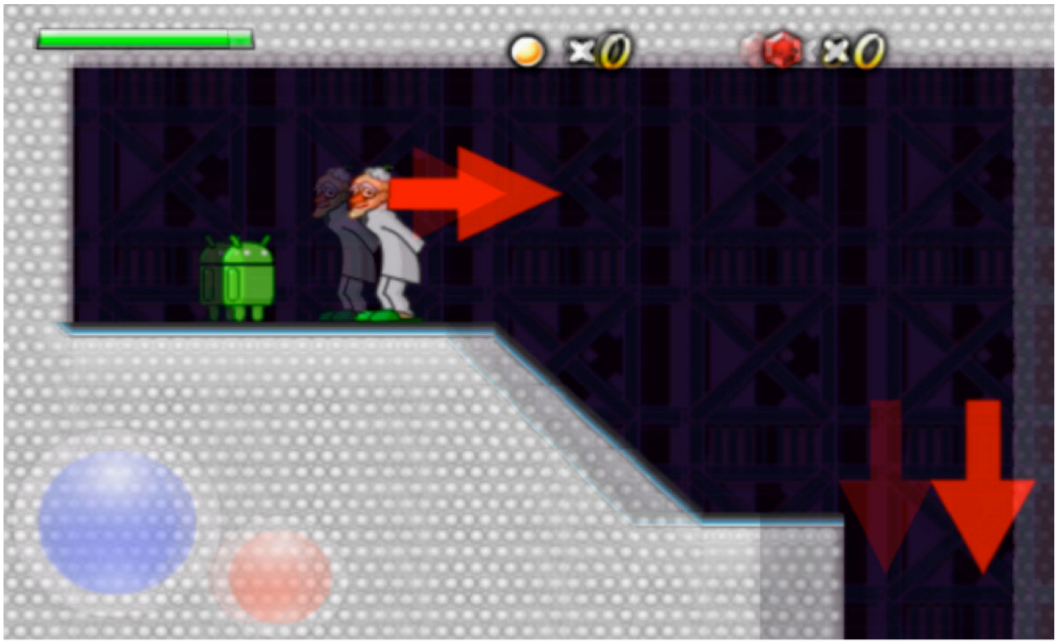

Figure 5–3. *Replica Island stretched from 480×320 to 800×480 pixels, overlaid with a faint image of how it is actually rendered on a 800×480-pixel display*

Replica Island does something very intelligent here: it performs normal stretching on the y-axis with the scaling factor we just calculated (1.5). But instead of using the x-axis scaling factor (1.66), which would make the image look squished, it uses the y-axis scaling factor. This neat little trick allows all objects on the screen keep their aspect ratio. A 32! 32-pixel sprite becomes 48! 48 pixels instead of 53! 48 pixels. However, this also means that our coordinate system is no longer bounded between (0,0) and (479,319). Instead it now goes from (0,0) to (533,319). And this is why we see more of the world of Replica Island on a Nexus One than on an HTC Hero.

Note, however, that using this fancy method might not be appropriate for some games. For example, having the world size depend on the screen aspect ratio could give an unfair advantage to players with wider screens. This would be the case in a game like Starcraft 2. Also, if you want the entire game world to fit onto a single screen (like in Mr. Nom), it would be better to use the simpler, stretching method. With the fancier version, we'd have blank space left over on wider screens.

A Simpler Solution

Replica Island has one advantage: it does all this stretching and scaling via OpenGL ES, which is hardware accelerated. So far we've only discussed how to draw to a `Bitmap` and a `View` via the `Canvas` class, which doesn't involve hardware acceleration on the GPU, but slow number-crunching on the CPU.

We'll therefore perform a simple trick: we'll create a framebuffer in the form of a `Bitmap` instance that has our target resolution. This way we don't have to worry about the actual screen resolution when designing our graphical assets or when rendering them via code. We just pretend that the screen resolution is the same on all devices. All our draw calls will target this "virtual" framebuffer `Bitmap` via a `Canvas` instance. When we're done rendering a frame of our game, we'll simply draw this framebuffer `Bitmap` to our `SurfaceView` via a call to the `Canvas.drawBitmap()` method, which allows us to draw a `Bitmap` stretched.

If we want to use the same technique as Replica Island, we just need to adjust the size of our framebuffer on the bigger axis (i.e., on the x-axis in landscape mode, and on the y-axis in portrait mode). We also have to make sure that we fill the extra pixels we get so there's no blank space.

The Implementation

So let's summarize all this by forming a simple plan of attack:

- We design all our graphic assets for a fixed target resolution (320! 480 in Mr. Nom's case).
- We create a `Bitmap` the same size as our target resolution and direct all our drawing calls to it, effectively working in a fixed-coordinate system.
- When we are done drawing a frame of our game, we draw our framebuffer `Bitmap` stretched to the `SurfaceView`. On devices with a lower screen resolution the image will be scaled down, and on devices with a higher resolution it will be scaled up.
- We have to make sure that all the UI elements the user interacts with are big enough at all screen densities when we do our scaling trick. This is something we can do in the graphic asset–design phase using the sizes of actual devices in combination with the formulas shown previously.

Now that we know how we will handle different screen resolutions and densities, I can also explain the `scaleX` and `scaleY` variables we met when we implemented the `SingleTouchHandler` and `MultiTouchHandler` a few pages earlier.

All our game code will be tuned to work with our fixed target resolution (320! 480 pixels). If we receive touch events on a device that has a higher or lower resolution, the x- and y-coordinates of those events will be defined in the `View`'s coordinate system, not in our target resolution coordinate system. Thus we have to transform the coordinates from their original system to our system based on the scaling factors. Here's how we do that:

```
transformed touch x = real touch x * (target pixels on x axis / real pixels on x axis)
transformed touch y = real touch y * (target pixels on y axis / real pixels on y axis)
```

Let's calculate a simple example for a target resolution of 320! 480 pixels and a device with a resolution of 480! 800 pixels. If we touch the middle of the screen, we'll receive an event with the coordinates (240,400). Using the two preceding formulas, we arrive at the following, which is exactly in the middle of our target coordinate system:

```
transformed touch x = 240 * (320 / 480) = 160
transformed touch y = 400 * (480 / 800) = 240
```

Let's do another one, assuming a real resolution of 240! 320, again touching the middle of the screen, at (120,160):

```
transformed touch x = 120 * (320 / 240) = 160
transformed touch y = 160 * (480 / 320) = 240
```

Hurray, it works in both directions. If we multiply the real touch event coordinates by the target factor divided by the real factor, we don't have to care about all this transforming in our actual game code. All the touch coordinates will be expressed in our fixed–target coordinate system.

With that out of our way, let's implement the last few classes of our game framework.

AndroidPixmap: Pixels for the People

According to the design of our `Pixmap` interface from Chapter 3, there's not much to implement. Listing 5–12 shows the code.

Listing 5–12. *AndroidPixmap.java, a Pixmap Implementation Wrapping a Bitmap*

```
package com.badlogic.androidgames.framework.impl;

import android.graphics.Bitmap;

import com.badlogic.androidgames.framework.Graphics.PixmapFormat;
import com.badlogic.androidgames.framework.Pixmap;

public class AndroidPixmap implements Pixmap {
    Bitmap bitmap;
    PixmapFormat format;

    public AndroidPixmap(Bitmap bitmap, PixmapFormat format) {
        this.bitmap = bitmap;
        this.format = format;
    }

    @Override
    public int getWidth() {
        return bitmap.getWidth();
    }

    @Override
    public int getHeight() {
        return bitmap.getHeight();
    }

    @Override
    public PixmapFormat getFormat() {
        return format;
    }

    @Override
    public void dispose() {
        bitmap.recycle();
    }
}
```

All we do is store the Bitmap instance that we wrap, along with its format, which is stored as a PixmapFormat enumeration value, as defined in Chapter 3. Additionally we implement the required methods of the Pixmap interface so we can query the width and height of the Pixmap and its format, and also ensure that the pixels can get dumped from RAM. Note that the bitmap member is package private, so we can access it in AndroidGraphics, which we'll implement now.

AndroidGraphics: Serving Our Drawing Needs

The Graphics interface we designed in Chapter 3 is also pretty lean and mean. It will draw pixels, lines, rectangles, and Pixmaps to the framebuffer. As discussed, we'll use a Bitmap as our framebuffer and direct all drawing calls to it via a Canvas. It is also responsible for creating Pixmap instances from asset files. We'll thus also need an AssetManager again. Listing 5–13 shows the code for our implementation of that interface, AndroidGraphics.

Listing 5–12. *AndroidGraphics.java; Implementing the Graphics Interface*

```
package com.badlogic.androidgames.framework.impl;

import java.io.IOException;
import java.io.InputStream;

import android.content.res.AssetManager;
import android.graphics.Bitmap;
import android.graphics.Bitmap.Config;
import android.graphics.BitmapFactory;
import android.graphics.BitmapFactory.Options;
import android.graphics.Canvas;
import android.graphics.Paint;
import android.graphics.Paint.Style;
import android.graphics.Rect;

import com.badlogic.androidgames.framework.Graphics;
import com.badlogic.androidgames.framework.Pixmap;

public class AndroidGraphics implements Graphics {
    AssetManager assets;
    Bitmap frameBuffer;
    Canvas canvas;
    Paint paint;
    Rect srcRect = new Rect();
    Rect dstRect = new Rect();
```

The class implements the Graphics interface. It has an AssetManager member that we'll use to load Bitmap instances, a Bitmap member that represents our artificial framebuffer, a Canvas member that we'll use to draw to the artificial framebuffer, a Paint we need for drawing, and two Rect members we'll need for implementing the AndroidGraphics.drawPixmap() methods. These last three members are there so that we don't have to create new instances of these classes on every draw call. That would make the garbage collector run wild.

```
    public AndroidGraphics(AssetManager assets, Bitmap frameBuffer) {
        this.assets = assets;
        this.frameBuffer = frameBuffer;
        this.canvas = new Canvas(frameBuffer);
        this.paint = new Paint();
    }
```

In the constructor we get an AssetManager and Bitmap representing our artificial framebuffer from the outside. We store these in the respective members and additionally create the Canvas instance that will draw to the artificial framebuffer, as well as the Paint, which we'll use for some of the drawing methods.

```
    @Override
    public Pixmap newPixmap(String fileName, PixmapFormat format) {
        Config config = null;
        if (format == PixmapFormat.RGB565)
            config = Config.RGB_565;
        else if (format == PixmapFormat.ARGB4444)
            config = Config.ARGB_4444;
        else
            config = Config.ARGB_8888;

        Options options = new Options();
        options.inPreferredConfig = config;

        InputStream in = null;
        Bitmap bitmap = null;
        try {
            in = assets.open(fileName);
            bitmap = BitmapFactory.decodeStream(in);
            if (bitmap == null)
```

```
                throw new RuntimeException("Couldn't load bitmap from asset '"
                        + fileName + "'");
        } catch (IOException e) {
            throw new RuntimeException("Couldn't load bitmap from asset '"
                    + fileName + "'");
        } finally {
            if (in != null) {
                try {
                    in.close();
                } catch (IOException e) {
                }
            }
        }

        if (bitmap.getConfig() == Config.RGB_565)
            format = PixmapFormat.RGB565;
        else if (bitmap.getConfig() == Config.ARGB_4444)
            format = PixmapFormat.ARGB4444;
        else
            format = PixmapFormat.ARGB8888;

        return new AndroidPixmap(bitmap, format);
    }
```

The newPixmap() method tries to load a Bitmap from an asset file, using the PixmapFormat specified. We start off by translating the PixmapFormat into one of the constants of the Android Config class we used in Chapter 4. Next we create a new Options instance and set our preferred color format. We then try to load the Bitmap from the asset via the BitmapFactory. We throw a RuntimeException if something goes wrong. Otherwise we check what format the BitmapFactory decided to load the Bitmap with and translate that into a PixmapFormat enumeration value. Remember that the BitmapFactory might decide to ignore our desired color format, so we have to check afterward what it decoded the image to. Finally we construct a new AndroidBitmap instance based on the Bitmap we loaded and its PixmapFormat, and return it to the caller.

```
    @Override
    public void clear(int color) {
        canvas.drawRGB((color & 0xff0000) >> 16, (color & 0xff00) >> 8,
                (color & 0xff));
    }
```

The clear() method simply extracts the red, green, and blue components of the specified 32-bit ARGB color parameter and calls the Canvas.drawRGB() method, which will clear our artificial framebuffer with that color. This method ignores any alpha value of the specified color, so we don't have to extract it.

```
    @Override
    public void drawPixel(int x, int y, int color) {
        paint.setColor(color);
        canvas.drawPoint(x, y, paint);
    }
```

The `drawPixel()` method draws a pixel to our artificial framebuffer via the `Canvas.drawPoint()` method. We first set the color of our `paint` member variable and pass that to the drawing method in addition to the x- and y-coordinates of the pixel.

```
@Override
public void drawLine(int x, int y, int x2, int y2, int color) {
    paint.setColor(color);
    canvas.drawLine(x, y, x2, y2, paint);
}
```

The `drawLine()` method draws the given line to the artificial framebuffer, again using the `paint` member to specify the color when calling the `Canvas.drawLine()` method.

```
@Override
public void drawRect(int x, int y, int width, int height, int color) {
    paint.setColor(color);
    paint.setStyle(Style.FILL);
    canvas.drawRect(x, y, x + width - 1, y + width - 1, paint);
}
```

The `drawRect()` method first sets the `Paint` member's color and style attributes so that we can draw a filled, colored rectangle. In the actual `Canvas.drawRect()` call, we then have to transform the `x`, `y`, `width`, and `height` parameters to the coordinates of the top-left and bottom-right corners of the rectangle. For the top-left corner we simply use the x and y parameters. For the bottom-right-corner coordinates, we add the width and height to x and y and subtract 1. For example, imagine if we were to render a rectangle with an x and y of (10,10) and a width and height of 2 and 2. If we don't subtract 1, the resulting rectangle on the screen would be 3! 3 pixels in size.

```
@Override
public void drawPixmap(Pixmap pixmap, int x, int y, int srcX, int srcY,
        int srcWidth, int srcHeight) {
    srcRect.left = srcX;
    srcRect.top = srcY;
    srcRect.right = srcX + srcWidth - 1;
    srcRect.bottom = srcY + srcHeight - 1;

    dstRect.left = x;
    dstRect.top = y;
    dstRect.right = x + srcWidth - 1;
    dstRect.bottom = y + srcHeight - 1;

    canvas.drawBitmap(((AndroidPixmap) pixmap).bitmap, srcRect, dstRect,
            null);
}
```

The `drawPixmap()` method, which allows drawing a portion of a `Pixmap`, first sets up the source and destination `Rect` members that get used in the actual drawing call. As with drawing a rectangle, we have to translate the x- and y-coordinates together with the width and height to the top-left and bottom-right corners. We again have to subtract 1, or else we'll overshoot by 1 pixel. Next we perform the actual drawing via the `Canvas.drawBitmap()` method, which will automatically do blending as well if the `Pixmap` we draw has a `PixmapFormat.ARGB4444` or `PixmapFormat.ARGB8888` color depth. Note that we have to cast the `Pixmap` parameter to an `AndroidPixmap` in order to be able to fetch the bitmap member for drawing with the Canvas. That's a little bit nasty, but we can be sure that the Pixmap instance passed in is actually an AndroidPixmap.

```
@Override
    public void drawPixmap(Pixmap pixmap, int x, int y) {
        canvas.drawBitmap(((AndroidPixmap)pixmap).bitmap, x, y, null);
    }
```

The second drawPixmap() method just draws the complete Pixmap to the artificial framebuffer at the given coordinates. We again do some casting to get to the Bitmap member of the AndroidPixmap.

```
@Override
    public int getWidth() {
        return frameBuffer.getWidth();
    }

    @Override
    public int getHeight() {
        return frameBuffer.getHeight();
    }
}
```

Finally we have the methods getWidth() and getHeight(), which simply return the size of the artificial framebuffer the AndroidGraphics instance stores and renders to internally.

There's one more class we need to implement related to graphics: AndroidFastRenderView.

AndroidFastRenderView: Loop, Strech, Loop, Stretch

The name of this class should already give away what lies ahead. In the last chapter we discussed using a SurfaceView to perform continuous rendering in a separate thread that could also house our game's main loop. We developed a very simple class called FastRenderView, which derived from the SurfaceView class, we made sure we play nice with the activity life cycle, and we set up a thread in which we constantly rendered to the SurfaceView via a Canvas.

We'll reuse this FastRenderView class and augment it to do a few more things:

- It will keep a reference to a Game instance from which it can get the active Screen. We will constantly call the Screen.update() and Screen.present() methods from within the FastRenderView thread.
- It will keep track of the delta time between frames that gets passed to the active Screen.
- It will take the artificial framebuffer that the AndroidGraphics instance draws to and draw it to the SurfaceView, scaled if necessary.

Listing 5–13 shows the implementation of the AndroidFastRenderView class.

Listing 5–13. *AndroidFastRenderView.java, a Threaded SurfaceView Executing Our Game Code*

```
package com.badlogic.androidgames.framework.impl;

import android.graphics.Bitmap;
import android.graphics.Canvas;
import android.graphics.Rect;
import android.view.SurfaceHolder;
import android.view.SurfaceView;

public class AndroidFastRenderView extends SurfaceView implements Runnable {
    AndroidGame game;
    Bitmap framebuffer;
    Thread renderThread = null;
    SurfaceHolder holder;
    volatile boolean running = false;
```

This should look very familiar. We just need to add two more members: an AndroidGame instance and a Bitmap instance representing our artificial framebuffer. The other members are the same as in our FastRenderView from Chapter 3.

```
    public AndroidFastRenderView(AndroidGame game, Bitmap framebuffer) {
        super(game);
        this.game = game;
        this.framebuffer = framebuffer;
        this.holder = getHolder();
    }
```

In the constructor we simply call the base class's constructor with the AndroidGame parameter (which is an Activity; more on that in a bit) and store the parameters in the respective members. We also get a SurfaceHolder again, as we did previously.

```
    public void resume() {
        running = true;
        renderThread = new Thread(this);
        renderThread.start();
    }
```

The resume() method is an exact copy of the FastRenderView.resume() method, so we don't need to go over that again. It just makes sure that our thread plays nice with the activity life cycle.

```
    public void run() {
        Rect dstRect = new Rect();
        long startTime = System.nanoTime();
        while(running) {
            if(!holder.getSurface().isValid())
                continue;

            float deltaTime = (System.nanoTime()-startTime) / 1000000000.0f;
            startTime = System.nanoTime();

            game.getCurrentScreen().update(deltaTime);
            game.getCurrentScreen().present(deltaTime);

            Canvas canvas = holder.lockCanvas();
            canvas.getClipBounds(dstRect);
            canvas.drawBitmap(framebuffer, null, dstRect, null);
            holder.unlockCanvasAndPost(canvas);
        }
    }
```

The `run()` method has a few more bells and whistles. The first addition is the tracking of the delta time between each frame. We use `System.nanoTime()` for this, which returns the current time in nanoseconds as a long.

NOTE: A nanosecond is one-billionth of a second.

In each loop iteration, we start off by taking the difference between the last loop iteration's start time and the current time. To make working with that delta time easier, we convert it to seconds. Next we save the current time stamp, which we'll use in the next loop iteration to calculate the next delta time. With the delta time at hand, we call the current `Screen`'s `update()` and `present()` methods, which will update the game logic and render things to the artificial framebuffer. Finally we get ahold of the `Canvas` for the `SurfaceView` and draw the artificial framebuffer. The scaling is performed automatically in case the destination rectangle we pass to the `Canvas.drawBitmap()` method is smaller or bigger than the framebuffer.

Note that we've used a shortcut here to get a destination rectangle that stretches over the whole `SurfaceView` via the `Canvas.getClipBounds()` method. It will set the `top` and `left` members of `dstRect` to 0 and 0, and the `bottom` and `right` members to the actual screen dimensions (480! 800 in portrait mode on a Nexus One). The rest of the method is exactly the same as what we had in our `FastRenderView` test. It just makes sure that the thread stops when the activity is paused or destroyed.

```
public void pause() {
        running = false;
        while(true) {
            try {
                renderThread.join();
                break;
            } catch (InterruptedException e) {
                // retry
            }
        }
    }
}
```

The last method of this class, `pause()`, is again exactly the same as the `FastRenderView.pause()` method. It simply terminates the rendering/main loop thread and waits for it to completely die before returning.

We are nearly done with our framework. The last piece of the puzzle is the implementation of the `Game` interface.

AndroidGame: Tying Everything Together

Our little game development framework is nearly complete. All we need to do is tie together the loose ends by implementating the `Game` interface we designed in Chapter 3, using the classes we created in the previous sections of this chapter. Here's a list of responsibilities:

- Perform window management. In our context, that means setting up an activity and an `AndroidFastRenderView`, and handling the activity life cycle in a clean way.
- Use and manage a `WakeLock` so that the screen does not get dimmed.
- Instantiate and hand out references to `Graphics`, `Audio`, `FileIO`, and `Input` to interested parties.
- Manage `Screens` and integrate them with the activity life cycle.

Our general goal is it to have a single class called `AndroidGame` from which we can derive. All we want to do is implement the `Game.getStartScreen()` method later on to start off our game, like this:

```
public class MrNom extends AndroidGame {
    @Override
    public Screen getStartScreen() {
        return new MainMenu(this);
    }
}
```

I hope you can see why it pays off to design a nice little framework before diving headfirst into programming the actual game. We can reuse this framework for all future games that are not to graphically intensive. So let's discuss Listing 5–14, which shows the `AndroidGame` class.

Listing 5–14. *AndroidGame.java; Tying Everything Together*

```
package com.badlogic.androidgames.framework.impl;

import android.app.Activity;
import android.content.Context;
import android.content.res.Configuration;
import android.graphics.Bitmap;
import android.graphics.Bitmap.Config;
import android.os.Bundle;
import android.os.PowerManager;
import android.os.PowerManager.WakeLock;
import android.view.Window;
import android.view.WindowManager;

import com.badlogic.androidgames.framework.Audio;
import com.badlogic.androidgames.framework.FileIO;
import com.badlogic.androidgames.framework.Game;
import com.badlogic.androidgames.framework.Graphics;
import com.badlogic.androidgames.framework.Input;
import com.badlogic.androidgames.framework.Screen;

public abstract class AndroidGame extends Activity implements Game {
    AndroidFastRenderView renderView;
    Graphics graphics;
    Audio audio;
    Input input;
    FileIO fileIO;
    Screen screen;
    WakeLock wakeLock;
```

The class definition starts off by letting `AndroidGame` extend the `Activity` class and implement the `Game` interface. Next we define a couple of members that should be familiar. The first member is the `AndroidFastRenderView`, which we'll draw to, and which will manage our main loop thread for us. The `Graphics`, `Audio`, `Input`, and `FileIO` members will be set to instances of `AndroidGraphics`, `AndroidAudio`, `AndroidInput`, and `AndroidFileIO`—no big surprise there. The next member holds the currently active `Screen`. Finally there's a member that holds a `WakeLock`, which we'll use to keep the screen from dimming.

```
@Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        requestWindowFeature(Window.FEATURE_NO_TITLE);
        getWindow().setFlags(WindowManager.LayoutParams.FLAG_FULLSCREEN,
                WindowManager.LayoutParams.FLAG_FULLSCREEN);

        boolean isLandscape = getResources().getConfiguration().orientation ==
Configuration.ORIENTATION_LANDSCAPE;
        int frameBufferWidth = isLandscape ? 480 : 320;
        int frameBufferHeight = isLandscape ? 320 : 480;
        Bitmap frameBuffer = Bitmap.createBitmap(frameBufferWidth,
                frameBufferHeight, Config.RGB_565);

        float scaleX = (float) frameBufferWidth
                / getWindowManager().getDefaultDisplay().getWidth();
        float scaleY = (float) frameBufferHeight
                / getWindowManager().getDefaultDisplay().getHeight();

        renderView = new AndroidFastRenderView(this, frameBuffer);
        graphics = new AndroidGraphics(getAssets(), frameBuffer);
        fileIO = new AndroidFileIO(getAssets());
        audio = new AndroidAudio(this);
        input = new AndroidInput(this, renderView, scaleX, scaleY);
        screen = getStartScreen();
        setContentView(renderView);

        PowerManager powerManager = (PowerManager)
getSystemService(Context.POWER_SERVICE);
        wakeLock = powerManager.newWakeLock(PowerManager.FULL_WAKE_LOCK, "GLGame");
    }
```

The onCreate() method, which is the familiar startup method of the Activity class, starts off by calling the base class's onCreate() method, as it is required. Next we make the Activity full-screen, as we did in a couple of tests in the previous chapter already. In the next few lines we set up our artificial framebuffer. Depending on the orientation of the activity, we either want to use a 320! 480 framebuffer (portrait mode) or a 480! 320 framebuffer (landscape mode). To determine what screen orientation the Activity uses, we fetch the orientation member from a class called Configuration, which we get via a call to getResources().getConfiguration(). Based on the value of that member, we then set the framebuffer size and instantiate a Bitmap, which we'll hand to the AndroidFastRenderView and AndroidGraphics instances a little later.

NOTE: The Bitmap instance has an RGB565 color format. This way we don't waste memory, and all our drawing is a little faster.

We also calculate the scaleX and scaleY values that the SingleTouchHandler and MultiTouchHandler classes will use to transform the touch event coordinates to our fixed-coordinate system.

Next we instantiate the AndroidFastRenderView, AndroidGraphics, AndroidAudio, AndroidInput, and AndroidFileIO with the necessary constructor arguments. Finally we call the getStartScreen() method, which our actual game will implement, and set the AndroidFastRenderView as the content view of the Activity. All these helper classes we just instantiated will do some more work in the background, of course. The AndroidInput class will tell the touch handler it selected to hook up with the AndroidFastRenderView, for example.

```
@Override
public void onResume() {
    super.onResume();
    wakeLock.acquire();
    screen.resume();
    renderView.resume();
}
```

Next up is the onResume() method of the Activity class, which we override. As usual, the first thing we do is call the superclass method because we are good citizens in Android land. Next we acquire the WakeLock and make sure the current Screen gets informed of the fact that the game (and thereby the activity) has just been resumed. Finally we tell the AndroidFastRenderView to resume the rendering thread, which will also kick off our game's main loop, in which we tell the current Screen to update and present itself in each iteration.

```
@Override
public void onPause() {
    super.onPause();
    wakeLock.release();
    renderView.pause();
    screen.pause();

    if (isFinishing())
```

```
            screen.dispose();
    }
```

The onPause() method first calls the superclass method again. Next it releases the WakeLock and makes sure that the rendering thread is terminated. If we didn't terminate the thread before calling the current Screen's onPause(), we could run into concurrency issues since the UI thread and the main loop thread would both access the Screen at the same time. Once we are sure the main loop thread is no longer alive, we tell the current Screen that it should pause itself. In case the Activity is going to be destroyed, we also inform the Screen of that event so it can do any cleanup work necessary.

```
    @Override
    public Input getInput() {
        return input;
    }

    @Override
    public FileIO getFileIO() {
        return fileIO;
    }

    @Override
    public Graphics getGraphics() {
        return graphics;
    }

    @Override
    public Audio getAudio() {
        return audio;
    }
```

The getInput(), getFileIO(), getGraphics(), and getAudio() methods should need no explanation. We simply return the respective instances to the caller. The caller will always be one of our Screen implementations of our game later on.

```
    @Override
    public void setScreen(Screen screen) {
        if (screen == null)
            throw new IllegalArgumentException("Screen must not be null");

        this.screen.pause();
        this.screen.dispose();
        screen.resume();
        screen.update(0);
        this.screen = screen;
    }
```

The setScreen() method we inherit from the Game interface looks simple at first glance. We start off with some old-school null-checking, as we can't allow a null Screen. Next we tell the current Screen to pause and dispose of itself so it can make room for the new Screen. The new Screen is asked to resume itself and update itself once with a delta time of zero. Finally we set the Screen member to the new Screen.

Let's think about who will call this method, and when. When we designed Mr. Nom, we identified all the transitions between various `Screen` instances. We'll usually call the `AndroidGame.setScreen()` method in the `update()` method of one of these `Screen` instances.

Say we have a main menu `Screen` where we check if the Play button is pressed in the `update()` method. If that is the case, we'll want to transition to the next `Screen`, and we can do so by calling the `AndroidGame.setScreen()` method from within the `MainMenu.update()` method with a brand-new instance of that next `Screen`. The `MainMenu` screen will get back control after the call to `AndroidGame.setScreen()`, and should immediately return to the caller, as it is no longer the active `Screen`. In this case the caller is the `AndroidFastRenderView` in the main loop thread. If you check the portion of the main loop responsible for updating and rendering the active `Screen`, you'll see that the `update()` method would be called on the `MainMenu` class, but the `present()` method would be called on the new current `Screen`. This would be bad, as we defined the `Screen` interface in a way that guarantees that the `resume()` and `update()` methods will be called at least once before the `Screen` is asked to present itself. And that's why we call these two methods in the `AndroidGame.setScreen()` method on the new `Screen`. All is taken care of for us by the mighty `AndroidGame` class.

```
public Screen getCurrentScreen() {
        return screen;
    }
}
```

The last method is called `getCurrentScreen()`. It simply returns the currently active `Screen`.

We've now created an easy-to-use Android game development framework. All we need to do now is implement the `Screens` of our game. We can also reuse the framework for any future games we can think of, as long as they do not need immense graphics power. If we need that, we have to start using OpenGL ES. However, we only need to replace the graphics part of our framework for that. All the other classes for audio, input, and file I/O can be reused.

Summary

In this chapter, we implemented a full-fledged 2D Android game development framework from scratch that we can reuse for all our future games (as long as they are graphically modest). Great care was taken to achieve a good, extensible design. We could take this code we have and replace the rendering portions with OpenGL ES, making Mr. Nom go 3D.

With all this boilerplate code in place, let's concentrate on what we are here for: writing games!