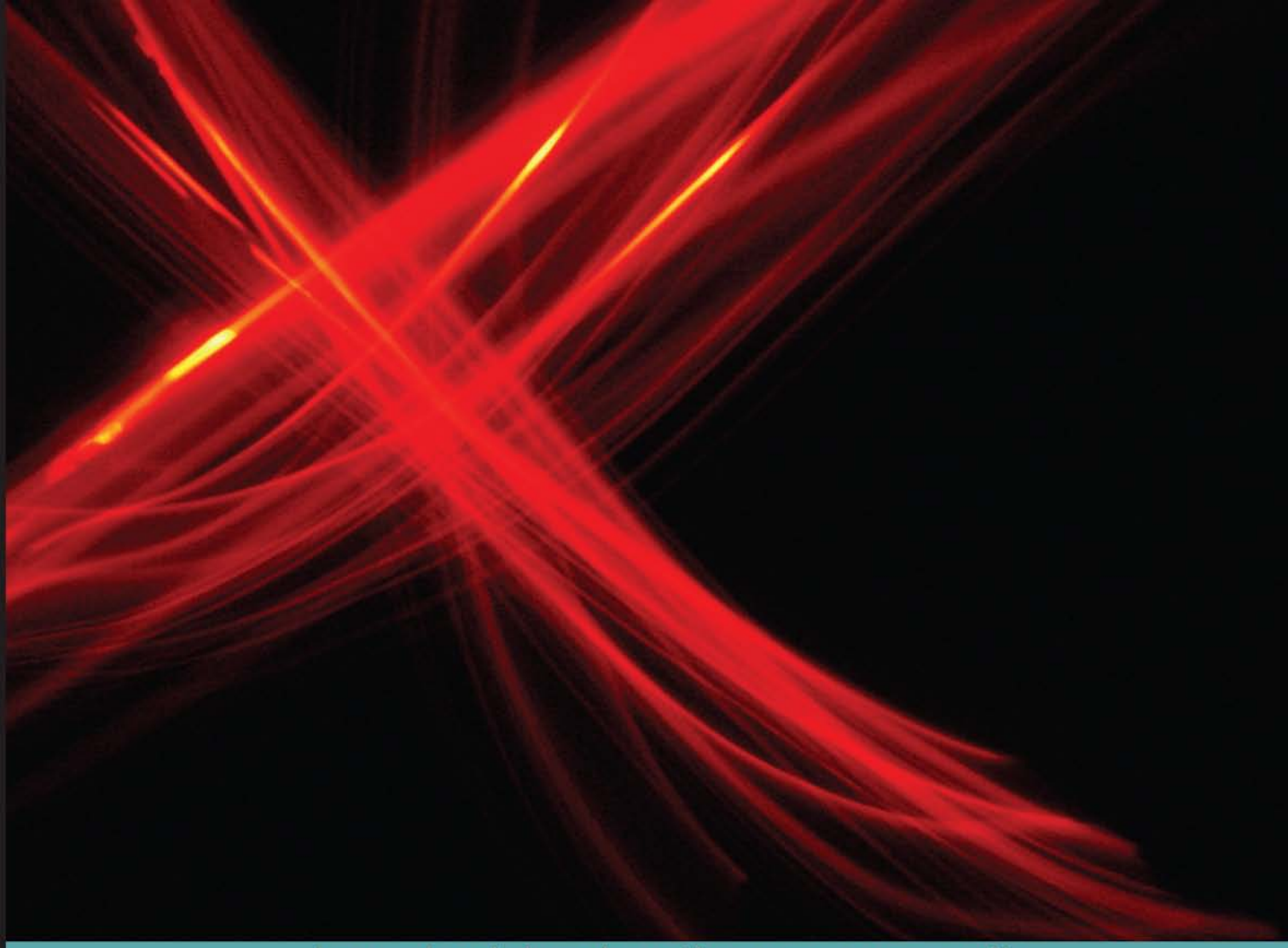


Learn by doing: less theory, more results

Android 3.0 Animations

Bring your Android applications to life with stunning animations

Beginner's Guide

Alex Shaw

[PACKT] PUBLISHING

Android 3.0 Animations

Beginner's Guide

Bring your Android applications to life with stunning animations

Alex Shaw

[PACKT] PUBLISHING

BIRMINGHAM - MUMBAI

Android 3.0 Animations
Beginner's Guide

First published: October 2011

Production Reference: 1211011

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-84951-528-3

www.packtpub.com

Cover Image by Vinayak Chittar (vinayak.chittar@gmail.com)

Credits

Author

Alex Shaw

Reviewers

Nathan Schwermann

Roger Belk

Acquisition Editor

Tarun Singh

Development Editors

Pallavi Iyengar

Meeta Rajani

Technical Editors

Lubna Shaikh

Ankita Shashi

Copy Editor

Leonard D'Silva

Project Coordinator

Shubhanjan Chatterjee

Proofreaders

Stephen Silk

Samantha Lyon

Indexer

Monica Ajmera

Graphics

Geetanjali Sawant

Production Coordinator

Melwyn D'sa

Cover Work

Melwyn D'sa

About the Author

Alex Shaw has been an Android fan boy since Android 1.5 arrived, and he began developing software for it almost immediately. He has presented at DroidCon in Berlin and London, and written applications for business, academia, and pleasure. An alumnus of The University of Edinburgh, he has kept close business and social ties with the Scottish geek scene. His consulting company, Glastonbridge Software Limited, provides development resources to the Edinburgh software industry. In his spare time, he writes generative music applications and talks a lot of nonsense.

Hearty thanks to my partner, Amy Worthington, for putting up with my constant stream of ideas whenever writing was on my mind. Thanks also to my Mum and to my close friends, who have supported me when stress and anxiety were taking their toll. Thanks also to the team at Packt, who put up with my erratic e-mail discipline and occasional late submissions, with patience and kindness. This book has been an adventure and an experience to remember.

About the Reviewers

Nathan Schwermann is a husband and proud father. He attends the University of Kansas to study Computer Science. In the past years, Nathan has worked as a freelance Android developer, making many great applications to help pay his high tuition costs. Nathan aspires to work for an independent gaming studio.

Roger Belk, also known as Big Daddy App, has developed Android applications for the last year. He is a self taught 43-year-old Ironworker. He builds applications using Eclipse with Android SDK, Java, and Google's App Inventor. You can check out his website at `www.BigDaddyApp.com`

Roger is also a power user in Google's App Inventor forums, answering help requests from new AI developers, from the setup, to the How-To, and to the coffee shop just chatting and kicking around ideals for AI apps.

Books that he has worked on include Animation 3.0 and Google App Inventor.

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit `www.PacktPub.com` for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`http://PacktLib.PacktPub.com`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface

Android 3.0 Animation, a Beginner's Guide, will introduce each of the most popular animation techniques to you as an Android developer. Using step-by-step instructions, you will learn how to create interactive dynamic forms, moving graphics, and 3D motion.

You will be taken on a journey from simple stop motion animations and fades through to moving input forms, and then on to 3D motion and game graphics. In this book we will create standalone animated graphics, three-dimensional lifts, fades, and spins. You will become adept at moving and transforming form data to bring boring old input forms and displays to life.

What this book covers

Chapter 1, *Animation Techniques on Android*, is a guided tour of the diverse possibilities for animating content on Android.

Chapter 2, *Frame Animations*, teaches you to create and control animations that are composed of a series of still images.

Chapter 3, *Tweening and Using Animators*, adds animated life to the Views in your Android application.

Chapter 4, *Animating Properties and Tweening Pages*, introduces some more specialized animation capabilities, available in Android.

Chapter 5, *Creating Classes for Tween Animation*, shows you how to take control of the low-level behaviors of your animations to create new and distinctive movements.

Chapter 6, *Using 3D Visual Techniques*, takes techniques that we introduced in previous chapters and shows you how to use them to create 3D depth and rotation effects.

Chapter 7, *2D Graphics with Surfaces*, introduces programmatic animations that you draw onto a blank canvas. This technique is ideal for writing games and advanced visualizations.

Chapter 8, *Live Wallpapers*, shows you how to build your animations into one of Android's most distinctive graphical features – wallpapers that move.

Chapter 9, *Practicing Good Practice and Style*, shows you how animation can be used to make your application better looking and easier to use, as well as looking at the performance cost of animated graphics.

What you need for this book

You should know how to program in Java and have experience using the Android SDK to make Android applications. You should understand basic object-oriented programming and know how to run your code on an Android device. You should also understand that Android uses XML files to show Views on screen.

You will require a computer that has the Android SDK installed and which has the Android 3.0 packages. You will also need a tool for entering the example code, compiling it, and deploying to an Android device or emulator. For this purpose, the book has been written with Eclipse users in mind, but the concepts and code presented will work equally well in IntelliJ IDEA or any other Android development environment that you are familiar with.

Because the Android applications in this book can be run on real devices, you may want to have an Android 3.0-compatible device, or higher. This is not necessary, but it is much more fun!

Who this book is for

If you are familiar with developing Android applications and want to bring your apps to life by adding smashing animations, then this book is for you. This book assumes that you are comfortable with Java development and have familiarity with creating Android Views in XML and Java. The tutorials assume that you will want to work with Eclipse, but you can work just as well with your preferred development tools.

Conventions

In this book, you will find several headings appearing frequently.

To give clear instructions of how to complete a procedure or task, we use:

Time for action – heading

1. Action 1
2. Action 2
3. Action 3

Instructions often need some extra explanation so that they make sense, so they are followed with:

What just happened?

This heading explains the working of tasks or instructions that you have just completed.

You will also find some other learning aids in the book, including:

Pop quiz – heading

These are short multiple choice questions intended to help you test your own understanding.

Have a go hero – heading

These set practical challenges and give you ideas for experimenting with what you have learned.

You will also find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "Get the example android project, `CountDroid`, from the code bundle and compile it to an Android APK for deploying to a device."

A block of code is set as follows:

```
package com.packt.animation.viewexample;
import android.app.Activity;
import android.os.Bundle;
public class ViewExample extends Activity {
  @Override public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    MyTextView helloView = new MyTextView(this);
    helloView.setText("Hello Views!");
    setContentView(helloView);
  }
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
  <ImageView
    android:id="@+id/stickman"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:src="@anim/stickman"
```

```
/>
<LinearLayout
  android:orientation="horizontal"
  android:layout_width="fill_parent"
  android:layout_height="wrap_content"
  android:gravity="center"
>
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: " Download the APK you built to your favorite emulator or Android device, and launch the **Counting Calculator** activity.".

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a book that you need and would like to see us publish, please send us a note in the **SUGGEST A TITLE** form on `www.packtpub.com` or e-mail `suggest@packtpub.com`.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.PacktPub.com`. If you purchased this book elsewhere, you can visit `http://www.PacktPub.com/support` and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/support`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
Animation Techniques on Android

There are so many ways to make an animation on Android that you might get a little lost if you don't know where to start. So here is the grand tour! You'll find out how all the techniques in this book can be used to make your application stand out from the rest.

Animations are divided loosely into a spectrum. At the top of the spectrum there are the simplest kinds of animation, and at the bottom there are the most complex kinds.

In this chapter, we shall look at the following:

- Animations that just show the same animation every time you play it. Or, if they do change, then they follow some simple pre-defined pattern. These are **frame** animations.
- Animations that apply to a widget-based application, which take an ordinary input form and move it around in a way that means something. These are the **tweens** and the **animators**.
- Animations that can show anything calculated on the fly from whatever data they are given. Games are made like this. These are the **surface-based** animations.

Android lets you combine these three techniques, but you get to choose them. Let me show you what I mean.

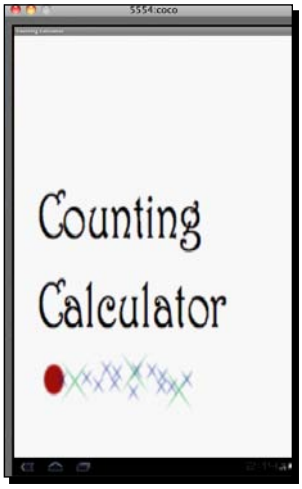
An animated application: counting calculator

Let's get started by presenting an example showing many different types of animation. **Counting calculator** is a very simple calculator application, which is designed to familiarize children with the idea of adding two numbers together. To make it easy-to-use, animated elements direct the child's attention to the things that happen as they use the calculator. And for slightly older users (that would be us), we can see how Android brings animation to an interactive application.

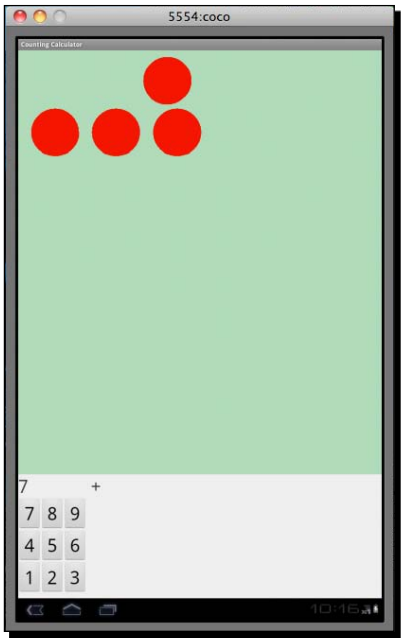
Time for action – learning to count with the counting calculator

We'll get started by trying out the counting calculator right now!

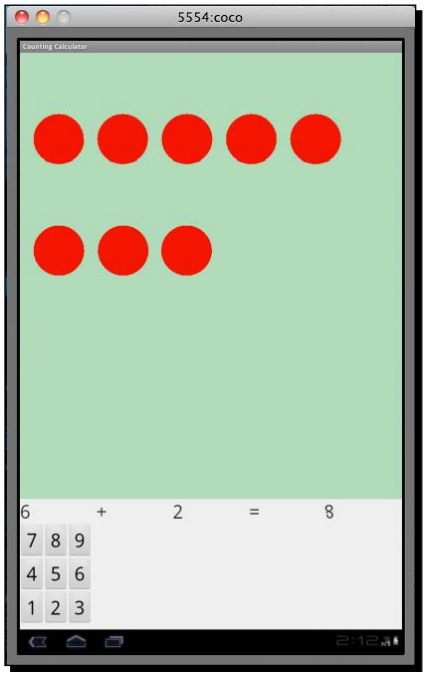
1. Get the example android project, `CountDroid`, from the code bundle and compile it to an Android APK for deploying to a device.
2. You can build it either straight from the command line using `Ant`, or import it into Eclipse using **Existing projects into workspace** from the **Import…** dialog.
3. Download the APK you built to your favorite emulator or Android device, and launch the **Counting Calculator** activity.
4. Wait for the calculator to appear. Notice that whenever something happens, it is animated:



5. Press a number key and watch what happens. What do you think will happen when you press another number?



6. Press another number and watch what happens. You'll see an animated visualization in the top-half of the display:



What just happened?

The first thing you'll see is the animated splash screen, which displays a short sequence introducing the application, and then you are delivered to the calculator itself. The splash screen is decorative; it's a little canned sequence to announce the purpose of the application in a fun way. The number buttons are animated, and so is the display. As you choose numbers, you'll see an animation on the top of the screen, counting the number of balls. There are several different sorts of animations, each expressing a different sort of meaning in the application.

Have a go hero – explore the counting calculator application

We're only just getting warmed up, but some people just have to see the code right away! If you are a fearless code explorer yourself, open up your favorite editor and have a look at the XML and Java elements of the **Counting Calculator** application. If you find anything confusing right now, don't panic! The animation concepts in the **Counting Calculator** will be explained throughout this book.

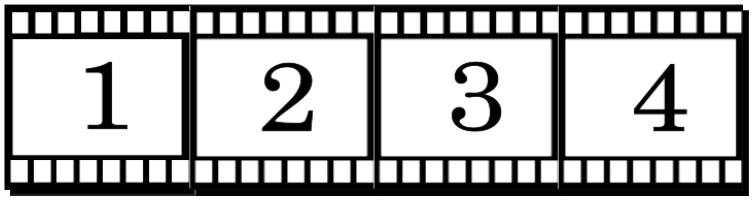
Here is a handful of clues to get you started:

- There is a simple animation in `res/drawable`, defined in XML. Look in this folder for the `splash.xml` file.
- There are a few small animations in the `res/anim` folder. They are used elsewhere in the application layout.
- There is a big complex animation in `BallField.java`.

See if you can work out which animations apply to which part of the user interface in the **Counting Calculator**.

Frame animation

The first sort of animation we'll look at is the animation that we used to make the splash screen. It's called a frame animation. Using this technique, you make an animation by creating several images and displaying them one after the other, like a strip of cinema film.



It's exactly like making a film from a reel of still cells, or drawing a series of pictures in the corner of a book and flicking through them. Each frame is shown for a very short time before moving on to the next one, so that the eye does not realize that the images are distinct.

Frame animations

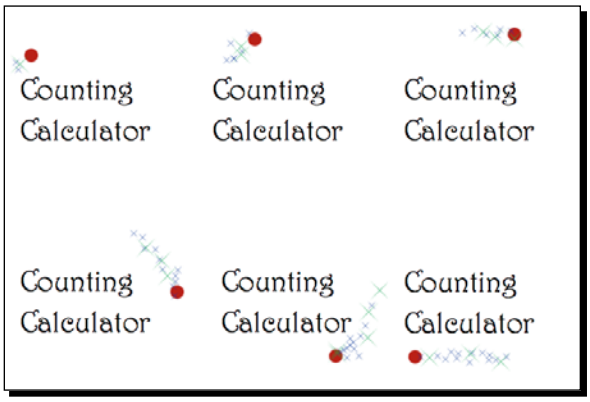
The term frame animation comes from movies, where it was considered that individual cells of a film resembled tiny framed pictures. Just remember that it works by showing a series of similar pictures, one after another. Because all animation works by making a series of updates to a picture, you will sometimes hear me using the word **frame** to describe other animation techniques. Don't be confused! When I am talking about Android frame animations, I will make sure I say the exact term "frame animation".

Time for action – playing with the frames

Time for some fun with frame animations! Let's have a play with the one in the splash screen of the **Counting Calculator**. Right now, we'll play around with changing the images; I'll show you the code part in the next chapter.

1. Open up your favorite image editor; anything that handles the PNG graphics format will do. If you have an editor that supports layered graphics, you might find it easier to load all the images as different layers, rather than loading every picture individually. Yours truly favors the GIMP for this purpose.

2. In your image editor, choose the option to open a file. Browse to the folder that contains the `CountDroid` project and navigate to the `res/drawable` directory from there. Select and load the images marked as `splash_*.png`.



In the previous screenshot, you can see a selection of frames taken from the **Counting Calculator** splash screen. Take a look at the images in order, and see how the images are all individual **moments** of the overall animation.

3. Try making some changes to the graphics. Add a dot on each image, in a slightly different place each time.

4. Save all of the images that you changed, so that they overwrite the existing images in the `res/drawable` folder.

5. If you are using Eclipse, refresh the `res/drawable` folder by selecting it in the **Package Explorer** and pressing *F5*. This will ensure it knows that it needs to recompile your APK.

6. Go back to your `CountDroid` project and rebuild it. Download the new version of the application to your device.

7. Load up the new application and watch its splash screen.

8. Go back to step four and play around! Get used to making things happen, by changing them incrementally. Can you make the dot appear to move in a straight line?

What just happened

You've just seen where the true work of the splash animation is done—we drew it! The animation we made is quite jerky, and that is because it is being redrawn every `100` milliseconds. For totally smooth playback of animation, the human eye needs to be shown a new image at least once every `42` milliseconds.

At faster speeds, you need to make more frames for your animation, or they will finish too soon. If you go too fast, the graphics in most mobile devices will not be able to keep up and will **drop** frames. And the human eye will not notice the difference anyway.

`100` milliseconds are enough for us to see an animation without getting too bored making it, although it does look rather jerky for now.

You now know how to make things move around by changing frames, and this is exactly how professional animators all over the world do it (well, maybe you need a few more tools to make a Hollywood blockbuster, but the principle is the same).

This technique is ideal when:

- Making animated graphics such as progress spinners and eye-catching game avatars
- Re-using existing animated graphics, for instance, animations that have been prepared for the web.

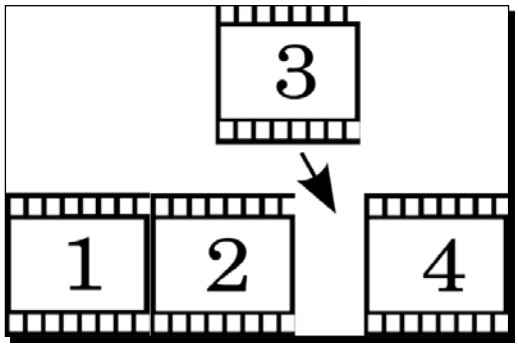
- Showing that something is happening behind the scenes, such as a file copying widget that shows files moving to their destination.
- Keeping it simple! It's a simple process, and if you work with non-technical designers and artists, they will be able to understand it easily.

Android provides an API for creating these sorts of animations in XML, and I'll show you how to make your own in *Chapter 2*, *Frame Animations*.

Fancy frame animations

You might be thinking that frame animations are just for playing back pre-made animations, but that's not quite the whole story. Since Android is a bit more advanced than ordinary paper, you can do a lot more with it.

- You can apply playback controls to a frame animation by using Java. This means that you can pause the animation, speed it up, slow it down, and so on, as your application changes.
- You can add images and you can remove images from the animation. As they are graphical elements, the frames of the animation use the same graphics scaling techniques that other static graphics use. Let's say you use a lot of animations having only a couple of frames that are different. You could implement them all as only one animation, and add the right frames to it when you show it to the user.



So although your pictures are drawn and added into the application as static elements, you can still make your animations interactive by choosing what you want to show next.

Simple fades using transition animations

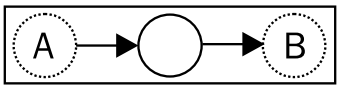
Sometimes you don't want to make a complex animation using lots of frames, but you would like to transition smoothly between two drawables without having to provide the in-between graphics as separate images.

If you are in this position, you can use transition animations. Transition drawables are another type of drawable element that does simple fades between two images. Although the idea of them is a little bit different to frame animation, you will find that they work in a similar way. I'll demonstrate this to you in *Chapter 2*, *Frame Animation*.

Tweening

Tweening is short for in-betweening. It provides the vital link between two fixed points or key frames.

Android provides a simple way to declare an animation that you can apply to your existing application. This time around, you don't need to add any extra graphics as you did with the XML animation. Instead, you use the views and widgets that are already in your application, making your widget-based interface itself represent dynamic content. You can move views around, distort them, make them vanish, or appear in all sorts of interesting ways. Google likes to call these animations tweens, taking a graphical element between one state and the next.



A simple tween takes the display from one place to another. In the preceding diagram, the centre ball represents the tween.

To implement tweening, Android uses a neat little class called, unsurprisingly, `Animation`. By using this class and its subclasses, you can not only write great XML animations, but as it keeps this sort of thing neatly encapsulated, your Java will look clean too. You can pick from several pre-made animations or write your own. Let's drill down into what we can do with tween animations.

Time for action – finding tweens

Let's take a look around the **Counting Calculator** application, and see where tweens have been used.

1. Open up the **Counting Calculator** on your Android device.
2. As you use the application, look for animations that might be using the `Animation` class.

Finding the tween animations in an application

All animations apply to views, so look for things that might be implemented using a standard `View` class.

What just happened?

You probably saw that the display and the buttons were just simple `TextViews` and numbers, and that's exactly right. When they move around, it's all thanks to the `Animation` class. `Animation` classes were chosen for this purpose, because I wanted to use the Android views system to display information. There are other ways to display text and write buttons that you'll see later, but views are simple and consistent.

A chess game might move between two different squares on the board, but if the piece just vanishes and reappears, then it just looks jerky. A tweening animation would show the piece sliding to its new destination.

A networked application might take a few seconds to fetch a bit of data. You can represent that transition behind the scenes with an appropriate animation on its screen.

The tween jazz band

Translations are an animation that moves a view from one place to another. This is great for showing where things are going, such as songs being added to a playlist, or deleted items flying into the trash can. All of the tween animations that you will have seen in the **Counting Calculator** make use of translations.

Alpha animations, as their name implies, can shift the alpha value of a view from invisible to translucent to solid, or vice versa. By fading view elements gently in and out, you can change the focal point of your application in a smooth way.

Rotate animations, well, you should be able to guess what this does.

Scale animations make things bigger and smaller. This can be used effectively to add a 3D feel, without having to mess around with the mathematics of true 3D.

Interpolations—meet the drummer

Every animation needs an interpolator. Interpolators are separate classes that control animations, by telling them how fast they should be doing things. Modulating the speed of the animation as it goes along makes the character of the animation change.

There are several built-in interpolator types that completely change the character of an animation. For instance, an **accelerate interpolator** makes the view accelerate to its destination. On the other hand, a **bounce interpolator** would make an animation bounce back-and-forth as it reaches its destination, giving the feeling of a ball coming to a rest. The bounce interpolator has a much more playful character, and is therefore more suited to fun applications than to an application that should look businesslike.

You can think of interpolations as being the drummer in the band, as they control the rhythm of the animation.

Animation sets—meet the conductor

If an interpolation is the drummer, an animation set is a conductor. Using an animation set consisting of several different animations, you can make more interesting things happen to a view.

You can combine animations at the same time, for instance, a rotating-scaling animation might make an object appear to corkscrew into or out of the screen.

You can also combine animations in sequence, for instance, a translate tween followed by a scale tween would make an object appear to move, reach its destination, and then get bigger or smaller.

You can even combine animation sets.

Tweening elements in XML

You can apply a transition animation in several places in your code. In fact, if you're writing in Java, you can set one off at any point. But Android provides entry points for animations in a few places, which are listed as follows:

- When the window first appears on screen, an animation can be used to show the view being put in place. This can also be used whenever the screen changes, for instance, due to portrait-landscape rotation.
- When you interact with some widgets, you can call an animation to make the interaction smoother.
- When you change pages on a book-style application, you can animate the switch to the next page.

What are tweens good at?

You're probably getting a feel for how you're expected to use animations by now. To sum up, they're suitable for use in situations like the following:

- Working with existing display elements
- Displaying secondary data to give the user information about the transition behind the scenes
- Encapsulating transforms on display elements, not just for XML but also to keep your Java code tidy too

Animators – new in Android 3.0!

As you are no doubt aware that views in Android GUIs have many accessors defining their position, their color, and so on, wouldn't it just be simpler to animate a view by changing those parameters a little bit each frame? This feature has been introduced in Honeycomb; it's called an **Animator**.

At its most basic level, animators are little daemon threads that wake up, change a view a bit, and go back to sleep till the next frame. They are a lot like the old `Animation` classes for tweening, but they are more generalized.

For instance, an Animator would allow you to modify the background color of a view, something that no tween could do. And if you have implemented your own views with special properties, they will work for those too.

However, unlike a tween, they are not designed to go between two states. If you want that sort of functionality, you will have to program it yourself. They are also less descriptive to use in your code—a tween allows you to say "*translate this object* `200` *pixels on the* `X-axis`", but an Animator says "*increment the* `X` *parameter by* `200` *pixels*".

I will show you how they compare to tweens in more detail in *Chapter 3*, *Tweens and Animators*.

Pop quiz – view animations and drawable animations

Okay, it's time for you to see what you've learned so far! We've talked about two different sorts of animation: the ones you can use with pictures and the ones that you can use with view elements. There are a lot of ideas floating around; what can you remember?

1. What is a transition drawable?
    a. A view that switches between two views
    b. A drawable that switched between two drawables
    c. A view that switches between two drawables
2. You would like a tween to move faster towards the end of its animation. What parameter of the tween would you use to do this?
    a. Its interpolator
    b. Its grandmother
    c. Its animator
3. You have a set of PNG images that you want to combine into an animation. You would use…
    a. An animator
    b. An interpolator
    c. A frame animation
4. Which of the following animates views moving from one place to another?
    a. An animator
    b. A slider
    c. A tween
5. In the counting calculator, which of these animations is least likely to use a tween?
    a. The bouncing balls
    b. The equation display
    c. The keypad

Beyond views: high-speed animating in 2 dimensions

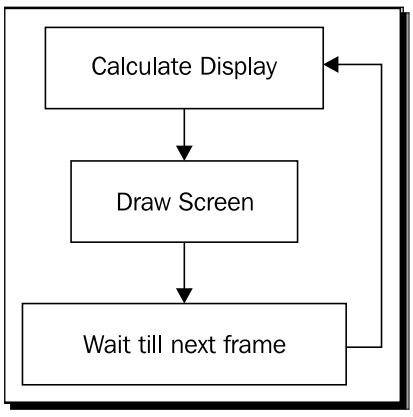
This is where things get really exciting! Sometimes you want to draw things to a screen and have full control over how they're drawn. This is especially true for animated elements, where you might want to draw some visualizations in real time. For instance, a graphic equalizer that moves in time with a playing MP3. You could draw it completely using views, but it would be horribly slow, and your phone would drain its battery much faster. This is because views do lots of extra work that makes them excellent for general interaction, like drawing themselves, handling clicks correctly, and so on. But that's not always useful, and it takes time. If only you could tell Android that you want to reserve an area of your screen for drawing freely, where you can look after the redrawing yourself, and optimize it for whatever your application does.

Well don't worry, because you can do exactly that! Surfaces are raw areas of screen that you can draw whatever you like into. You can refresh them when you want, handle user touches however you like, and you even get a handy toolkit of common vector and bitmap drawing operations to make use of.

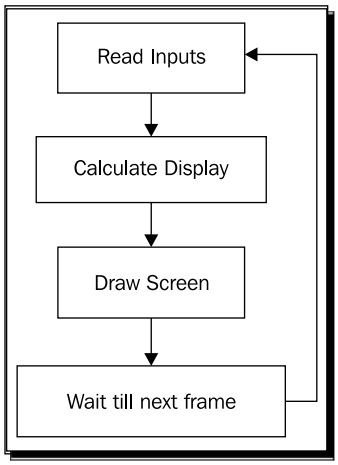
If frame animations are like films, surface-based animations are like clockwork toys. Each thing has to be told programmatically how it should appear on the screen. Look at the bouncing balls in the `Counting Calculator` example; each one of those is drawn by a **bounce** routine and a **count** routine that manipulates everything about them.

Drawing loops

One thing that you will need to do when you are drawing animations like this is to keep updating the image with the next piece of the animation. Usually, you will find that you want to make incremental changes to your animation model at regular intervals, and then update the screen to represent it. This is a common pattern, especially in game programming. In your code, you make a loop that looks like the following screenshot:



This is what your loop will look like, if there are no external interactions with your animation or if the calculations in **Calculate Display** are modified top down by a supervising object (as it was for the ball animation in the **Counting Calculator**). But if you are writing something that handles user interactions in its own way, you will want to add in a separate stage to handle this.



This is commonly called a **game loop** in games programming, and forms the basis for most applications that involve game-like interaction between a computer and a user. You will find that writing animation loops and game loops is in itself fairly simple, but you'll need to make sure that you write one when you're using a surface to handle your animations.

Doing your own housekeeping is hard

If you wanted to rewrite your entire application on a surface, you'd probably get something up and running pretty quickly, but you'd then spend weeks trying to reproduce all of the nice features that your `TextViews` and `ListViews` gave you. So don't do this as it's not worth it (I've tried), just take it from me.

Where to use surfaces

You can use surfaces in the following scenarios:

- When you want to compute the appearance of your animation, rather than follow a predefined pattern
- When you are making something that doesn't follow the widget-based interaction model
- When you are making something that needs to run fast
- Games
- Live wallpapers
- DSP and music visualizations (maybe that's just me)

What do views do anyway?

How should you decide that views are too complex and you should use a surface? As a simple example, we'll look at how views are drawn to the screen. It will help us see a little more into what views do behind the scenes, and go some way to explain why views are better for form data and why surfaces are fast.

We will take a look at a slightly modified version of the **Hello World** application that the Eclipse ADT plugin generates for you, automatically. I recommend that you use Eclipse, because it provides you with an integrated suite of debugging tools, and because it helps you navigate your code easily.

When you have a complex form, the routines we will see being called in the example would be triggered a lot, and for many view objects.

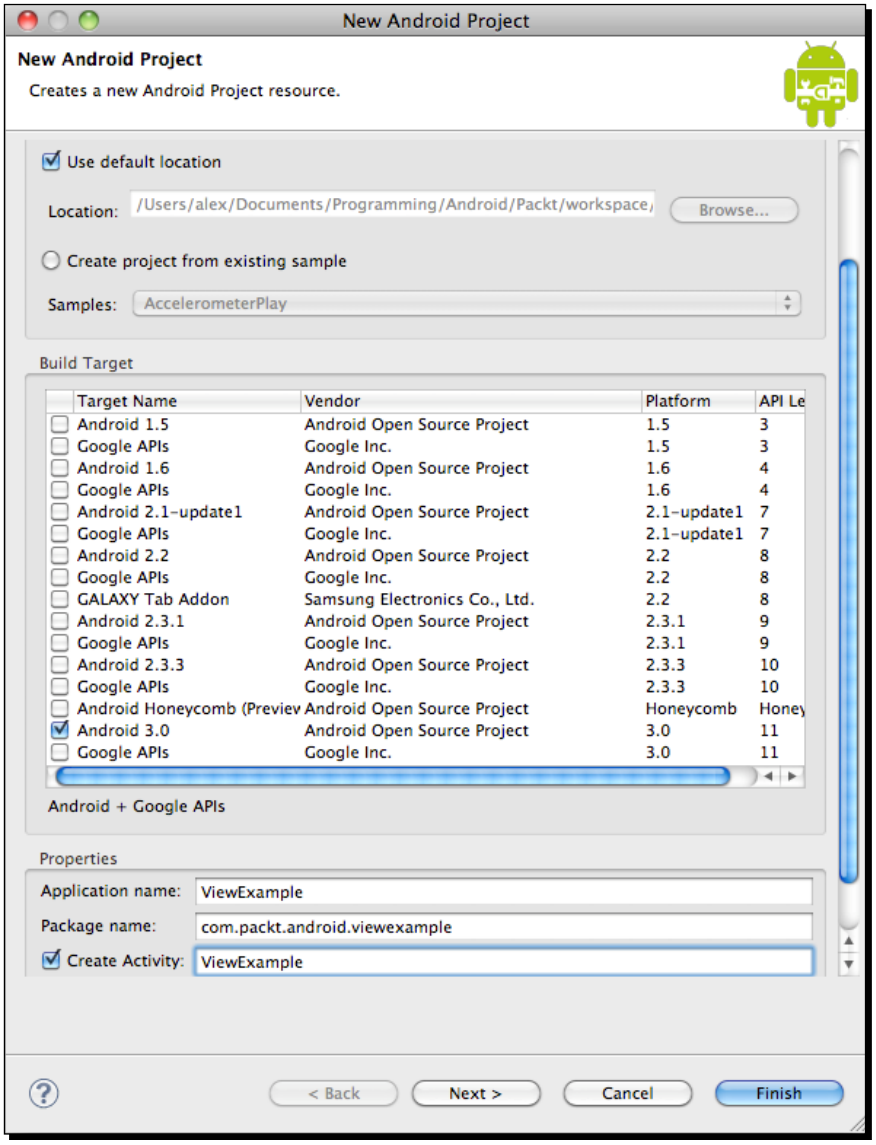
Time for action – let's draw views

For this example, we will create our own `TextView` class and breakpoint into it. Fortunately, for us, we will be able to base it on the `android.view.TextView` class, so the amount of code we need to write is minimal.

1. Create a new Android project in Eclipse. I will describe this particular example in Eclipse, because it provides easy access to the Java debugger. However, you can substitute your preferred development tool, if you are comfortable with using breakpoints in it.

2. On the **New Android Project** page, enter the following settings and hit the **Finish** button:

   - **Project Name:** `ViewExample`
   - **Build Target:** `Android 3.0`
   - **Package Name:** `com.packt.animation.viewexample`
   - **Create Activity:** `ViewExample`

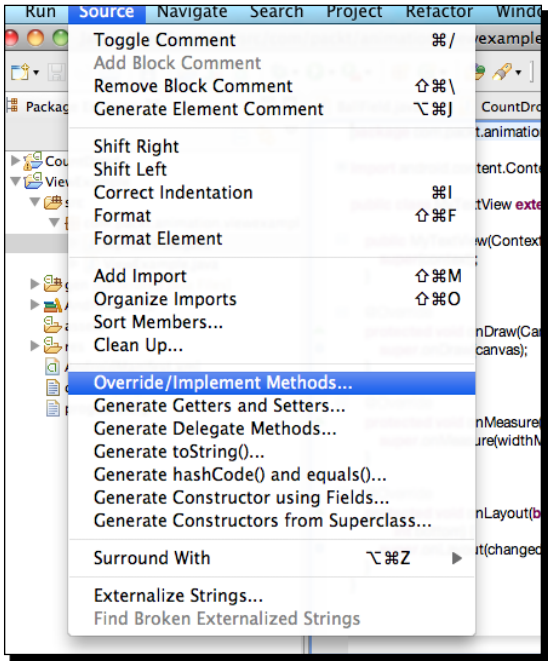


3. Create a new class in the `com.packt.animation.viewexample` package, called `MyTextView`. It should override `android.widget.TextView`, and implement a constructor `MyTextView(Context)`. Your class should look like this:

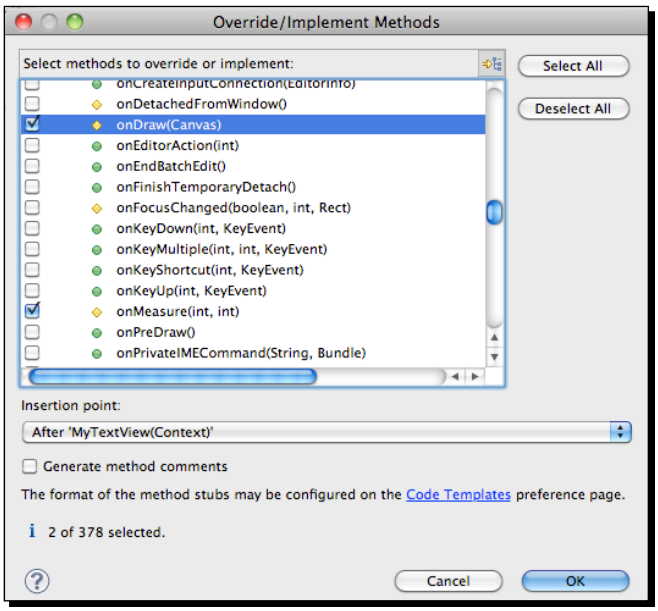
```
package com.packt.animation.viewexample;

import android.content.Context;
import android.widget.TextView;

public class MyTextView extends TextView {
  public MyTextView(Context context) {
    super(context);
  }
}
```

You can download the example code files for all Packt books you have purchased from your account at `http://www.PacktPub.com`. If you purchased this book elsewhere, you can visit `http://www.PacktPub.com/support and register` to have the files e-mailed directly to you.

4. Click the **Source** menu and choose **Override/Implement Methods...**.



5. There are a lot of methods there, aren't there?



They represent functionality that the views system provides for free. (If you're not using Eclipse, skip to *point 7*).

6. Select the **onDraw(Canvas)**, **onLayout**, and **onMeasure(int, int)** methods (you'll need to expand the view's parent class to see **onLayout**) and continue.

7. You will need to add one new constructor to the class. It will end up looking like the following block of code:

```
package com.packt.animation.viewexample;
import android.content.Context;
import android.graphics.Canvas;
import android.widget.TextView;
public class MyTextView extends TextView {
  public MyTextView(Context context) {
    super(context);
```

```
  }
  @Override protected void onDraw(Canvas canvas) {
    super.onDraw(canvas);
  }
  @Override protected void onMeasure(int widthMeasureSpec, int
    heightMeasureSpec) {
    super.onMeasure(widthMeasureSpec, heightMeasureSpec);
  }
  @Override protected void onLayout(boolean changed, int left, int
    top, int right, int bottom) {
    super.onLayout(changed, left, top, right, bottom);
  }
}
```

Here we have subclassed the `TextView` class, but we haven't changed any behavior in it. We have overridden some of the Java methods in order to make it easy to interrupt the flow of the program and see how it is working.

8. You will see that there are some `super()` method calls in the generated code. Left-click in the margin next to each one, and select **Toggle Breakpoint**, so that there is a breakpoint next to each of them.

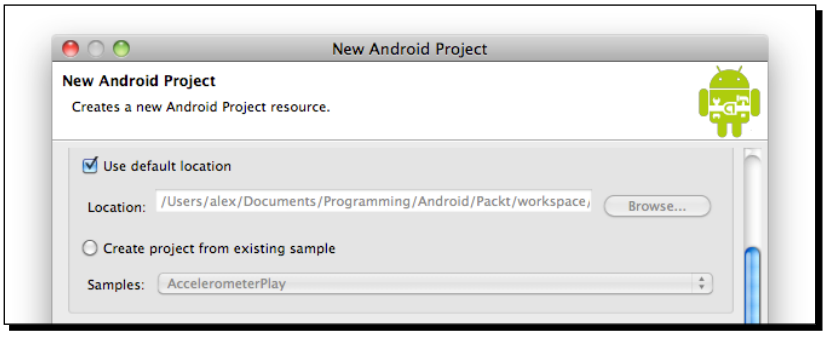
9. Next, edit `ViewExample.java` (also in `com.packt.animation.viewexample`) so that it is using `MyTextView` instead of `TextView`. It should look like the following block of code:

```
package com.packt.animation.viewexample;
import android.app.Activity;
import android.os.Bundle;
public class ViewExample extends Activity {
  @Override public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    MyTextView helloView = new MyTextView(this);
    helloView.setText("Hello Views!");
    setContentView(helloView);
  }
}
```

Here we are simply calling our new `MyTextView`, instead of the one that is used in the default **Hello Android** application.

10. Navigate to **Debug**, in the **Run** menu, to debug this application in an emulator or an attached device. As it loads, you will see that it calls all of the methods, which we have break-pointed.

    - It calls `onMeasure` to evaluate how much space it needs to allow `MyTextView` on the screen
    - It calls `onLayout` to tell the widget how much space it has been allocated and to place the widget onto a specific location on the screen



    - It calls `onDraw` last to actually place text on screen

11. Try rotating the device, or changing the orientation of the emulator, by holding down the *Ctrl* key and pressing *F11*, and watching it go through the layout process again.

What just happened?

You've seen just a snippet of the useful features that the views system provides. This is just for one simple widget; you can imagine how much work Android does when there is a more complex form element like a `ListView` on the screen. By using widgets, you don't have to worry about what happens when you rotate your screen. Widgets also handle interaction for you, and they provide features such as cursor navigation where you can use a directional joypad to navigate the screen rather than a touch screen.

You've also seen that there are a lot of optional features that you can use, if you wish. If you use a surface, you will still need to provide a basic set of layout and drawing operations in order for your animations to appear onscreen, but you can also consolidate your program logic. Instead of doing separate redraw and layout operations for each and every view, you can produce your own lightweight version.

But remember that there is a cost; you will lose a lot of functionality that could be important to you. Only use a surface when you are sure that your animation will not be required to have features that a form would have, such as text input, or when you are prepared to write all the extra functionality yourself.

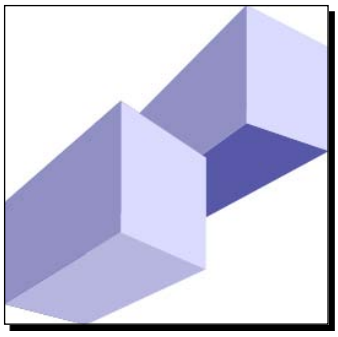
In *Chapter 7*, *2D Graphics with Surfaces*, you will learn a little more about this. Android does still give surface users access to some view elements through the `SurfaceView` class, but it is still very much up to you to handle basic interaction.

Animating awesome 3D

You can also animate three-dimensional graphics in Android. There is a selection of common routines that developers use when drawing 3D objects. So long as you understand, loosely, how they work, there's no need to get a degree in mathematics to use them.

In *Chapter 7*, *2D Graphics with Surfaces*, I will give you a few stock routines that can be used and combined to make elemental 3D scenes. I will show what works with surfaces and what works with views; in general, there is quite a lot you can do with both.

The following screenshot is an example of three-dimensional cubes, with shaded sides and vanishing points:



3D is all about mathematics, but Android does give you a few tools to make it somewhat easier to get that elusive third dimension into your applications. In fact, you can get some cool 3D effects using the techniques I've already told you about.

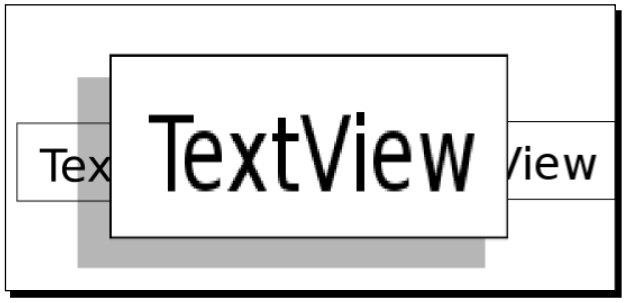
It's not all about the mathematics though. As my old computer graphics lecturer once told me, "*nobody cares if your 3D is realistic, as long as it looks good*", or something like that. I was a student at the time and probably not paying very much attention, but that's beside the point. 3D is all about perception.

Simple effects can give your animations the feeling of real depth, even better than a computed 3D world. Sometimes this is a preferable technique when writing applications that must run on slow Android devices. This becomes even more important for us, because animations need to be re-calculated regularly enough to look smooth. Also, a simpler pseudo-3D style provides a cleaner visual experience than what comes with true three-dimensional visualizations.

For instance, all of the following can give a sensation of three-dimensional depth, without having to do very much math at all.

- Drop shadows
- Color and alpha changes
- Scale changes

In the following screenshot, you can see a simple illustration of using drop shadows and scale to give a sense of depth, without all the complex messing around that true 3D techniques require:



Want to go faster?

Android supports a very fast 3D graphics system called **Open Graphics Libary for Embedded Systems** (**OpenGL ES**). When you see realistic 3D animations and games on Android, they are very likely to be using OpenGL ES. It is found on lots of operating systems, not just Android. It's maintained by the Khronos group, and has been available on Android since version 2.0 (Eclair).

Another, very fast way to generate complex graphics is to use native code, written in C or C++ programming languages. Native code can be faster than Java, because you can hand-optimize your code to make it faster. The Android **Native Development Kit** (**NDK**) supports this approach and can even be combined with OpenGL ES.

Unfortunately, learning how to use OpenGL ES would require too much theory to fit into a beginner's book, and learning C++ would take even longer still. However, once you have mastered the basics of animation on Android, you may want to consider this route to get fast, realistic animation into your application.

Have a go hero – what is right for your application

This section is for anyone who has an application right now that they want to add a bit of animated juice to, and for anyone who's planning on creating an application. Grab some paper, as this is going to be the planning stage where you can plan out the animations for your interface. You can refer to this section whenever you have an application that you think needs a little animated zest!

If you don't have any applications in mind right now, but you still want to be a hero, think of an application you feel would be cool to make.

1. Write down what the application does, and what concepts it involves. For instance, if you had an application that displays tide times for the local beach, the concepts might have been sea, tides, and scheduling.
2. Imagine how you'd expect a user to interact with your application. Get some ideas about what screens they might see.
3. Draw a screen from this application, as it would appear to the user.
4. Annotate each part of the screen that might have an animated component, saying what it is. Think about the concepts you wrote down in point 1, and how the animation is reflective of this. For instance, in the tide times application, it might be cool to have animations appear in waves.
5. Pick one of the animated components you made; we'll think about how we'll draw it next.
6. Would you want the animation to be pre-made with an image editor and drawn on the screen? If so, it sounds like you could use a frame animation.
7. Do you want to fade between two Drawable elements? A transition drawable is for you!
8. Would you want the animation to take place amongst the views and widgets on screen? Perhaps you could make use of a tween or an animator.

9. Do you need to do something that isn't really provided by the ordinary Android view system that suits vector drawing or high performance? Time for a surface-based animation.
10. Go back to point 3 and think about another animation that you might use in your application. Repeat until you're so excited that you just have to start coding.
11. Bonus question: Will it look exciting or will it just confuse the user?

With any luck, you've got a list of ideas to add animations to your application, and also what kind of technology it uses. The upshot of which is that you'll know exactly what you need to read next in order to perfect your application. The chapters are self-contained, so you can skip ahead, if you want to get stuck in! (However, if you want to take the traditional journey from beginning to end, that's fine by me.)

Making a better application

Remember who the audience for your application is and choose something that will satisfy them. I'll give you a load of good tips and advice in Chapter 9, *Good Practice and Style*, but here are couple of tips to get you started.

Always be helpful

It is all too easy to get carried away with lots of flashy graphics that only make your application more confusing. It is your responsibility whenever you give information to a user to draw their attention to the part that is most important. If there is a dancing amphibian (to pick an example at random) in one corner of the screen, they might not pay enough attention to whatever it is that you actually want them to notice. On the other hand, if the important data is presented with a bit of animated excitement, it will be impossible for them to miss it.

Small and powered by batteries

Throughout this book, it is worth remembering that you are writing for an operating system that is designed to be embedded in small, battery-powered devices. By allowing you to extend your information visualizations into the fourth dimension, animation can allow you to make more efficient use of a small screen. But animations are more power-hungry than other visual elements, and too much will flatten your user's battery. We will discuss this in detail in Chapter 9, *Good Practice and Style*. However, as a general guide, remember that fast graphics routines are also efficient in terms of power.

Pop quiz – putting it all together

1. Which of these gives an animation a 3D feel?
    a A slow transition between two frames
    b A change in scale
    c A shift from red to blue
2. How can you draw several objects straight to the screen without declaring views?
    a Use an animator
    b Use an interpolator
    c Use a surface
3. Which of the following manipulates views to make an animation?
    a An animator
    b A surface
    c OpenGL ES
4. Why would you want to use a surface?
    a For maximum performance and flexibility
    b To get more colors
    c To fade between two frames
5. Adding lots of animations is…
    a Always good
    b Good for battery life
    c Only useful if they improve user experience
6. You want to animate one of Android's built-in widgets on a loop, which is most likely to be the right thing to do?
    a Put an animator on the parameters to animate
    b Re-implement the widget using a surface
    c Use a tween and keep restarting it when it finishes

Summary

Now you know a little bit more about how the Android animation systems work, on their own as well as together in an application. You've seen how:

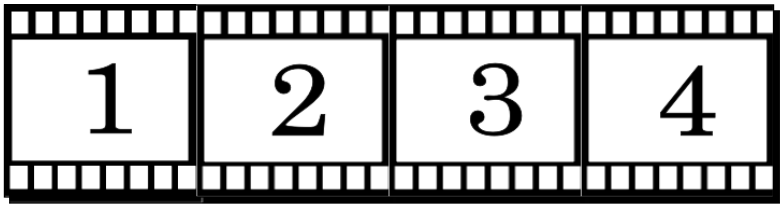
- A frame animation can be used to display animations composed of several pre-prepared frames and that its strength is its simplicity
- A transition is a simple fade between two animations
- Tweens bring motion to humdrum old form interfaces, explaining highlighting and events to make interaction intuitive and fun
- Animators let you animate a view, or any other compatible object, by making continuous modifications to its display parameters
- Surfaces allow you to strip away the ordinary Android display features, and get a bare-bones engine to show fast, efficient animations
- View-based animations are integrated more with the core functionality of Android, such as rotating screens and widget-based displays; surfaces are faster than views because they don't do any of this.
- Although OpenGL 3D is fast and flashy, you can create a lot of 3D graphics with a simple surface and some mathematics.

In the next chapter, we will learn about frame-based animations and create some frame animations of our own!

2
Frame Animations

Frame animations build up animation from a series of still images that are shown in rapid succession, like a film reel.

Each frame in a frame animation is shown for a fraction of a second and then it moves on to the next frame, giving an impression of motion like a film reel, as in the following graphic:



In this chapter, we shall:

- Make a frame animation in XML
- Use Java to change a frame animation
- Make a transition animation, a simpler relative of frame animation

A few sections of this will seem like revision if you are already comfortable with the Android views system, but for the most part I'll assume you are already comfortable with it. Ready? Let's go…

Making a frame animation

First up, let's make a small activity which will run our first frame animation. Like a lot of graphical components in Android, we can use XML to define it. In fact, for this simple activity, we don't need to write any Java at all.

Remember, a frame animation works by taking a series of still images and displaying them in a particular order. When making frame animations, the still images that you use will usually be numbered, so that you can tell which order to display them in.

Time for action – the funky stick man

In a recent survey of Android users, 90 percent of those who were asked said that they were very happy with the Android platform except for the fact that their phone did not contain an animation of a dancing stick man.

Right now, we are going to make this killer app, using a stick figure prepared by the legendary stick figure artist Alexandro Del Shaw.

1. Create a new Android project with the following settings:
   - **Project name:** `Funky Stick Man`
   - **Build Target:** `Android 3.0`
   - **Application name:** `Funky Stick Man`
   - **Package name:** `com.packt.animation.funky`
   - **Create Activity:** `FunkyActivity`

2. Now, let's get some Drawables to work with. **Drawable** is a generic interface for anything on Android that is used for drawing graphics to the screen. In this case we will be using PNG images.

   Copy the `res/drawable-*/` directories from the code bundle for this chapter. You'll find a `ZIP` file called `tutorial_images_1`. Unzip it somewhere local to copy it to your new project.
   - In Eclipse, right-click on the res/ folder and select Import…
   - Pick the option to import by File System, then click Next
   - Navigate to the 5283_examples directory
   - Import the Drawables to the res/drawable-hdpi, res/drawable-mdpi, and res/drawable-ldpi folders in your project.

3. Create the folder `anim/` inside the `res/` folder.

4. In the `anim/` folder, create a new XML file called `stickman.xml`.

5. Create the root XML node in the file, so that it looks like the following:

```
<?xml version="1.0" encoding="utf-8"?>
<animation-list
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:oneshot="false">
</animation-list>
```

   This is the standard root node for a frame animation. The `android:oneshot` attribute says whether to stop playing the animation after its first iteration. If `false`, it loops back to the beginning and plays indefinitely. We want the little fellow to keep dancing, so we choose `false`.

6. Take a look at the `res/drawable-hdpi` directory and observe the numbered animation frames:

   - `stickman_frame_01.png`
   - `stickman_frame_02.png`

   (In fact, the `res/drawable-mdpi` and `res/drawable-ldpi` folders contain the same images in different resolutions.)

7. For each frame, add an `<item>` tag in numeric order to the `stickman.xml`, as in the following:

```
<?xml version="1.0" encoding="utf-8"?>
<animation-list
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:oneshot="false">
  <item
    android:drawable="@drawable/stickman_frame_00"
    android:duration="83"
  />
  <item
    android:drawable="@drawable/stickman_frame_01"
    android:duration="83"
  />
</animation-list>
```

   The `android:drawables` are obviously the images in the `res/drawable-*` directories. The `android:duration` attribute is simply the number of milliseconds that each `<item>` will be shown before the animation moves on to the next `<item>`.

   There! You have finished your animation!

8. Now for the final task, to display it to the user. Open up `res/layout/main.xml` and change the XML, so that it points to our new animation rather than the boring old **"Hello Android!"** text.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  android:gravity="center"
  android:background="#FFFFFFFF">
  <ImageView
    android:id="@+id/stickman"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:src="@anim/stickman"
   />
```

9. At this point, build and run your animation. If you've typed everything in as described it, you will see a stick figure, but he will not be animated.

You have made your animation, and now it is time for you to get Android to play it, when the application loads.

10. Navigate your way to the `Activity` class in `src/com/packt/animation/funky/FunkyStickMan.java`. We're going to add some code to it.

I'll highlight the code that you need to add and break it down, so that we can look at it one piece at a time.

11. Firstly, let's add the classes we're going to be working with. I'll describe them briefly here. You will see how they are used shortly.

```
package com.packt.animation.funky;
import android.app.Activity;
import android.graphics.drawable.AnimationDrawable;
import android.widget.ImageView;
import android.os.Bundle;
```

`AnimationDrawable` is Android's Java representation of the animation-list that we created in *step 3***.**

`ImageView` will be used solely so that we can access the `AnimationDrawable` stored within the `ImageView` in `main.xml`**.**

The next step will be to retrieve the `AnimationDrawable` that we added in the `main.xml` layout. We will access it through the `onCreate()` method in `FunkyActivity`, so that we start the animation as soon as the application

is loaded.

```
public class FunkyActivity extends Activity {
  /** Called when the activity is first created. */
  @Override
    public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.main);
      final ImageView animImage = (ImageView)
      findViewById(R.id.stickman);
      final AnimationDrawable animDrawable =
      (AnimationDrawable) animImage.getDrawable();
```

The previous code should look familiar if you have done any serious work with views before. Firstly, the **content view**, which is the main view that gets displayed on the screen, is added by `setContentView`. We pass in the resource ID that relates to the `main.xml` file we've just edited, so we know that it will contain the graphics we just added.

`FindViewById` gets the container `ImageView`. From there we call `getDrawable()` to get the actual animation. We will assign it to a final object called `animDrawable`, because we're going to need it in an anonymous `Runnable` in just a second.

12. Sorry to interrupt, the source code for `onCreate` continues as follows:

```
animImage.post(
  new Runnable() {
    public void run() {
      animDrawable.start();
    }
  }
);
  }
}
```

13. And here is where we actually start the animation. The call to `animDrawable.start()` is where we start the animation running.

Why did we not just call the `animDrawable.start()` method from the `onCreate()` method? You must make the `animDrawable.start()` call from inside a method that will be called from the GUI thread. In this example, we use a `.post(Runnable r)` method to do exactly that.

14. Let's build and run the new activity again.

Before, the stick man was not moving at all, but now he should be dancing. Watch the little fellow go!

What just happened?

Here we made a purely graphical application based around a single animation, with very little programming needed. The Android XML resource system took care of all the hard work for us.

The anatomy of a frame animation

Look at the structure of the XML in `stickman.xml`. If you ignore the root node and concentrate on the items, then it is simply a list of things to show in the order that you want to show them. It's just like a program! And, just like a program, if you change the order of declaration, then the things will be shown in a different order too.

Animation XML files are kept in `res/anim` as opposed to `res/drawable`, but the objects that the Android build tools will make will still be a subclass of Drawable.

XML elements

Here is a quick reference to the XML elements we use to create animations. The main headings are tags, and the subheadings are attributes of those tags.

<animation-list>

<animation-list> is always the top-level element in a frame animation. The `animation-list` is an ordered container of Drawable items.

```
<animation-list
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:oneshot="false">
```

This tag should contain the following options:

xmlns:android

All top-level Android XML elements declare their namespace as `xmlns:android="http://schemas.android.com/apk/res/android"`

android:oneshot

`oneshot` is a Boolean flag to specify whether to stop after the first playback iteration or to keep on looping the animation.

<item>

Each frame in the list is described as an `item` that references a pre-compiled `Drawable`.

```
<item
  android:drawable="@drawable/stickman_frame_01"
  android:duration="83" />
```

Each `item` should specify these attributes:

android:drawable

The `android:drawable` defines a reference to a Drawable using Android's usual resource hierarchy, that is, if your Drawables are in `res/drawable` or `res/drawable-?dpi`, you would use `@drawable/your_drawable_name`. It has to be specified.

android:duration

Specifies how long to show this item for. The value is given as an integer number in terms of milliseconds.

Timing

Of course, in addition to determinig which image to show next, the `<item>` tags also determine how long the image is shown, in milliseconds. Traditionally, movies are shown at `24` frames per second, because at that speed, most people cannot see any flickering or visual glitches. On a mobile device, you can get away with far fewer frames - around, around `12` frames per second still looks nice and smooth.

$$Milliseconds\ per\ frame = \frac{1000}{Frames\ per\ second}$$

Substituting the value `12` for the frames per second gives us:

$$Milliseconds\ per\ frame = \frac{1000}{12} \approx 83$$

This is why the duration values are all set to `83` in the *time for action* section.

For a smooth animation, you could use anything between `30` and `100` milliseconds per frame. The advantage of a small number is that it really displays smoothly. Larger delays might look jerky for some graphics.

The advantages of a larger delay are that it makes the user's device work less than an animation with short delays, saving valuable battery life. It will also take up less space in the target device's memory as it requires fewer frames to be stored. It also allows the guy who makes your animations to work a little less, as he doesn't have to make as many pictures!

You can also reduce unnecessary updates by putting a long delay on some frames, rather than animating several frames where nothing much is happening. There is no need for all of your frames to have the same duration.

Images and Drawables

This animation was constructed from pre-drawn PNG images of a stick man, which were added to the Android project in the same manner that you would add any image resource.

It is important when using graphics like this that you ensure that the source Drawables are all the same size. This is common sense when you think about it; each image has to take up the same amount of screen space as the one before it, in order to maintain the illusion that it is just one moving image. Failure to do this may result in ugly animations that have flickering edges.

The `<item>` elements in the animation list refer to a set of `Drawables` that can be located by Android at compile time. While we did use a lot of PNG images, we didn't in fact, need to do so. Android would be just as happy if we had used any other sort of `Drawable` type.

Screen size

Different Android devices come in different sizes, and you may have to provide graphical elements that work on a whole range of different screens. Because the frame animation makes use of the `Drawable` class, we are immediately able to take advantage of the portability features that Drawables provide.

In this particular example, we used the de-facto Android way to show images of different sizes, which is to say that we provided HDPI, MDPI, and LDPI versions of all the PNG images. If a user has a small screen, then the LDPI graphics are used. If (like most Android 3.0 devices) the device has a large, high-resolution screen, then we want to increase the resolution of the image to take advantage of this fact.

Sometimes you run out of memory

If, like me, you like to have a few graphical elements on screen at once, you will occasionally discover that your little mobile phone is not quite as powerful as your desktop computer when it comes to showing animations.

Sometimes, your application will crash instead of showing you the animation you ask it for. Check `LogCat` for error messages that look similar to this:

```
java.lang.OutOfMemoryError : bitmap size exceeds VM budget
```

When this happens, you will need to reduce either the size of your animation, the number of frames, or both. Fortunately, most Android 3.0 devices have a bit more memory than their Android 1 and 2 counterparts, but this still happens occasionally.

LogCat is an Android debugging tool that shows debug messages from all parts of the Android system, including your application. You can use it by navigating to the **LogCat** tab in the **DDMS** view of Eclipse, or by typing `adb logcat` from the command line (see `http://developer.android.com/guide/developing/tools/logcat.html` for more information).

Pop quiz – making frame animations

1. If you change the order of the `<item>` elements in an animation, what happens?
    a. Not a lot
    b. The animation changes at random
    c. The animation changes, because the frames are shown in a new order

2. A frame animation in XML has to be composed of which of the following?
    a. PNG images
    b. Anything which Android can represent as a Drawable
    c. Animators
3. You will find XML for animation declared in which folder?
    a. `res/Drawable`
    b. `res/anim`
    c. `src/Drawable`
4. Which of the following sentences is not true?
    a. All `android:duration` values in an animation have to be the same
    b. Animations with longer durations save battery life
    c. Animations with too long durations will look jerky
5. What is the top-level XML element that holds a list of animation frames called?
    a. `android:animation-list`
    b. `android:frameList`
    c. `android:frames`

Have a go hero – improve your dancing

Of course, our little man isn't exactly doing the most exciting dance. Following is a choreographed dance that he can do to look a bit funkier:

1. Dance to the left three times
2. Do a jump
3. Dance to the right once
4. Pause for a second
5. Do a jump

Guess what? It's your job to help him learn it, or rather, it's your job to program him to do it. Think you know what to do? You only need to make changes to the contents of `stickman.xml`. Here are some tips:

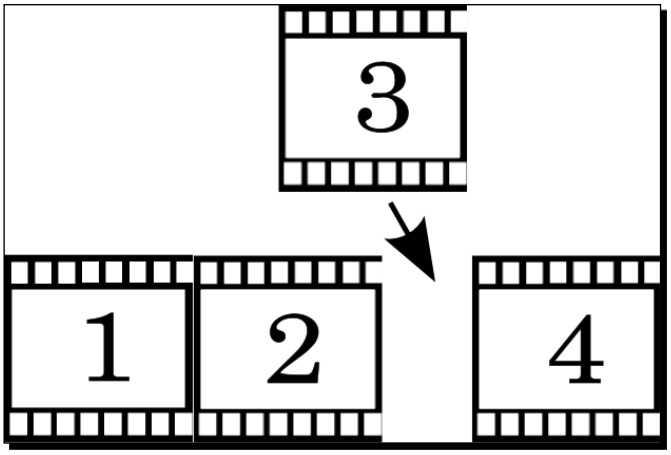
- Copying and pasting a block of `<item>` elements will cause that section of the animation to be replayed.

- Each dance move is exactly a second long, and aligned to the start of the animation image list. If you know how many frames there are in a second (read *Timing* again if you've forgotten), you can easily divide the animation into the chunks that you need.

Making frame animations in Java

XML animations are defined at build time, but what if you want to make an animation that is defined or modified at runtime?

We want to be able to change the animation by changing the animation sequence that is playing at any particular time.



As you have learned from the *Have A Go Hero – improve your dancing* section, you can affect all manner of changes to the character of an animation, just by moving a few frames around. You can use Java in your application to add dynamic changes to the animation, by directly manipulating the `AnimationDrawable` that defines the frame animation.

Time for action – making the stick man interactive

Our little man looks a bit lonely dancing by himself, doesn't he? Wouldn't it be nice if you could join in with the dancing? In this section, we will add buttons to the animation, to allow a user to change the dance moves that the stick man is doing.

In order to make the animation interactive, you're going to have to take the existing animation and split it up into its individual dance moves. There's a trick here: each dance move is exactly a second long. That means that the first `12` frames are the dance-left animation, the second `12` frames are the dance-right animation, and the final `12` define the jump animation.

We will make two new animations from the existing animation, one for dancing left and one for dancing right. The user interactions will be handled by changing which animation we are showing to the user.

Sounds simple? Let's go!

1. Open up the `Funky Stick Man` project you made in the first part of this chapter. Did you skip through to this part without doing the first? Well, if you think you're ready for it, you can find the first part already done in the code bundle.

   Unzip the project `5283_2_example1` from the code bundle for this chapter. Open it as you would normally open an Android project.

2. Open up `res/layout/main.xml` and add the following new buttons underneath the `ImageView` containing our stick man:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  android:gravity="center"
  android:background="#FFFFFFFF">
  <ImageView
    android:id="@+id/stickman"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:src="@anim/stickman"
  />
<LinearLayout
  android:orientation="horizontal"
  android:layout_width="fill_parent"
  android:layout_height="wrap_content"
  android:gravity="center"
>
  <Button
    android:id="@+id/danceleft"
    android:text="Dance Left"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
  />
  <Button
    android:id="@+id/danceright"
```

```
        android:text="Dance Right"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
       />
    </LinearLayout>
</LinearLayout>
```

These new buttons will be the controls that allow you to control the stick man's dance.

3. We want to divide our animation up into neat little parcels that can be triggered at a button press.

   Create two new animation XML files in `res/anim` called `dance_left.xml` and `dance_right.xml`. Give them both a top-level node of the type `<animation-list>`, as we did for `stickman.xml`.

   Set the `oneshot` attribute to `true`, as we don't want the animation to loop this time.

   When you're done, their XML should look like as follows:

```
?xml version="1.0" encoding="utf-8"?>
<animation-list
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:oneshot="true">
</animation-list>
```

4. Time to save us some typing! Go into `stickman.xml` and copy the first 13 items from the animation list. These images comprise the entire sequence required to show the stick man dancing to the left, plus one extra. Paste it into the middle of the new animation list in `dance_left.xml`.

The extra image (`stickman_frame_12`) is identical to frame `0`. It shows our funky stick man returning to a standing position. If we didn't do that, he would never quite finish his dance move and would look a bit awkward!

5. Repeat for `dance_right.xml` using the second 12 items.

```
<?xml version="1.0" encoding="utf-8"?>
<animation-list
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:oneshot="true">
<item
  android:drawable="@drawable/stickman_frame_12"
  android:duration="83"/>
```

```
<item
```

The first few lines of `dance_right.xml` should look like the previous code.

6. Now to wire them in to the buttons! Open up `com.pactk.animation.funky.FunkyActivity`. First we'll need to add a couple of new import declarations:

```
import android.view.View;
import android.widget.Button;
```

We're only importing these, so that we can wire our animation up to a button; we're not going to add any more animation code (yet!)

7. Add the following code to the end of the `onCreate` method in `FunkyActivity`:

```
Button danceLeftButton =
  (Button) findViewById(R.id.danceleft);
danceLeftButton.setOnClickListener(
  new View.OnClickListener(){
    public void onClick(View v) {
      // We'll fill this in in a minute
    }
  }
);
Button danceRightButton =
  (Button) findViewById(R.id.danceright);
danceRightButton.setOnClickListener(
  new View.OnClickListener(){
    public void onClick(View v) {
      // We'll fill this in in a minute too!
    }
  }
);
```

Just to get us started, we've defined two `onClick` interactions, to which we will add our animations.

8. Now to populate the `onClickListener` of the `danceLeftButton` with the actual animations! Let's start where we wrote *We'll fill this in in a minute*. We want to add three new lines of code to this method. I'll explain each one as we go along. Here's the first:

```
AnimationDrawable danceLeftAnim =
  (AnimationDrawable)
  getResources().getDrawable(R.anim.dance_left);
```

Here we are grabbing the `dance_left` animation that we defined in part 3 and filled out in part 4; we call `getResources.getDrawable(int id)` (a method of

the `Context` class) to get our animated Drawable, showing the stick figure dancing to the left.

Now, on to the next line…

```
            animImage.setImageDrawable(danceLeftAnim);
```

`animImage.setImageDrawable(Drawable d)` sets the animation as the active Drawable in our application's `ImageView`, replacing whatever animation is currently on the screen.

```
            danceLeftAnim.start();
```

Finally, we start the playing the animation here.

Now we need to do the same for the `danceRightAnim`. We'll add a similar piece of code where we promised *We'll fill this in in a minute too!*.

```
            AnimationDrawable danceRightAnim =
              (AnimationDrawable)
              getResources().getDrawable(R.anim.dance_right);
            animImage.setImageDrawable(danceRightAnim);
            danceRightAnim.start();
          }
        });
```

9. Time to build and test it.

What just happened?

You've made a reactive animation that's driven from a change in the application state. Or, in other words, you've made a stick man who dances when the user presses buttons!

By taking several animations and showing them to the user selectively, you made an animation that was dynamic and interactive.

Controlling frame animations

`AnimationDrawables` have a very simple interface, and there is not much you can do to manipulate the animation directly. These are some common methods you might want to call on your XML frame animations.

start() and stop()

Any questions about these two? Yes, that's right, you call them on an animation when you want it to start or stop. However, if you are using a one-shot animation, calling `start()` a second time will not take the animation back to the beginning. What can you do? Here's the answer…

AnimationDrawable.setVisible(true,true)

This is a very sneaky trick, so pay attention. The first parameter of `AnimationDrawable.setVisible` tells Android whether or not to make this Drawable visible, as you would expect.

The second parameter is subtler; if it is set to `true`, Android will reset your animation back to frame `0`. You will need to remember to do this, if you are re-using a one-shot animation in your user interface!

Creating new animations

You can't make changes to the frames and the order of an animation after it has been defined. But you can give the appearance of it changing by creating a new animation programmatically, with the new changes you want. Of course, it's perfectly simple to replace an old frame animation with a new one, created on the fly. Let's try making another little dance routine, but this time we will not define it in XML, but in Java.

Time for action – programmatically defined animation

Oh no! We forgot about one of the dance moves - we'd better add it back in! But you're probably getting a bit tired of defining animations in XML, so let's use a simple routine to generate the last animation when the `FunkyActivity` first loads.

We won't replace anything we wrote in the previous *Time For Action – making the stick man interactive*, but we'll add another button and associated animation.

1. Open up the `Funky Stick Man` Eclipse project again.
2. Open up `res/layout/main.xml` and add one more button:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  android:gravity="center"
  android:background="#FFFFFFFF">
  <ImageView
    android:id="@+id/stickman"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
  />
  <LinearLayout
    android:orientation="horizontal"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:gravity="center"
  >
    <Button
      android:id="@+id/danceleft"
      android:text="Dance Left"
      android:layout_width="wrap_content"
    android:layout_height="wrap_content"
     />
    <Button
      android:id="@+id/danceright"
      android:text="Dance Right"
      android:layout_width="wrap_content"
     android:layout_height="wrap_content"
    />
  </LinearLayout>
    <Button
      android:id="@+id/jump"
      android:text="Jump!"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
    />
```

```
</LinearLayout>
```

As you have no doubt guessed, this is the button we will use to show the jump animation. Our jump button is given the ID `jump`, and we will use this later.

3. We want to make a new animation, based on the last 12 frames of `@anim/stickman`. Rather than copy its XML, as we did for the `@anim/danceleft` and `@anim/danceright` animations, we will programmatically copy it from the existing `@anim/stickman` animation. In `FunkyActivity`, create a new method as shown in the following block of code. I'll talk you through it in just a minute.

```
private AnimationDrawable subAnimation(
  AnimationDrawable src, int start, int end ) {
    AnimationDrawable subAnim = new AnimationDrawable();
    subAnim.setOneShot( src.isOneShot() );
    for (int i = start; i<end; ++i) {
      subAnim.addFrame ( src.getFrame (i), src.getDuration (i)
        );
    }
    return subAnim;
  }
```

Now let's break that down (Don't type the same text in twice!).

```
private AnimationDrawable subAnimation(
  AnimationDrawable src, int start, int end ) {
    return subAnim;
  }
```

We create a general method for copying a subset of an animation to a new animation. It takes in three arguments - a source animation and the start and end of the range that you want to copy.

The method returns a new animation that only contains the frames between the specified start and end.

```
    AnimationDrawable subAnim = new AnimationDrawable();
```

As you can see, we construct an empty `AnimationDrawable` called `subAnim`.

```
    subAnim.setOneShot( src.isOneShot() );
```

It's nice and easy to copy across the `OneShot` value to our new animation, as it's just a Boolean. We'll want to change it later, but let's keep it here so that we can see we've copied everything.

```
    for (int i = start; i<end; ++i) {
      subAnim.addFrame ( src.getFrame (i), src.getDuration (i)
);
    }
```

We iterate over the `src` animation, copying out all frames between start and end. The call to add a frame to an animation is called `addFrame (`surprise surprise`)`, and it takes two arguments: the Drawable element to add, and the duration to show it for. In a way, it resembles the XML method of adding frames to an animation list.

(A truly reusable method would also contain error-handling methods, but I will leave this out for the sake of keeping this example clear.)

4. Next we want to use this method in the actual animation. Firstly, add a new member variable to the `FunkyActivity`:

```
private AnimationDrawable jumpAnim;
```

This will be our new animation for use with the **Jump!** button.

5. In the `onCreate()` method, append the following lines to the end:

```
jumpAnim = subAnimation( animDrawable, 24, 36 );
```

As you would expect, we initialize the `jumpAnim` using our new `subAnimation` method. We take all the animation frames from frame 24 (the 2-second mark) to frame 35 (there is an offset by one in effect in the copy operation).

```
Drawable standingStill =
  getResources()
  .getDrawable(R.drawable.stickman_frame_00);
```

Following on from that, you can see that we also append the very first stick man frame to the animation. This is done so that the stick man will return to a standing pose, once he has finished his little dance.

```
jumpAnim.addFrame(standingStill, 83);
jumpAnim.setOneShot(true);
```

Finally, we set the `OneShot` attribute to `true`. Recall from the XML tutorials that this means that we only show the animation once instead of looping.

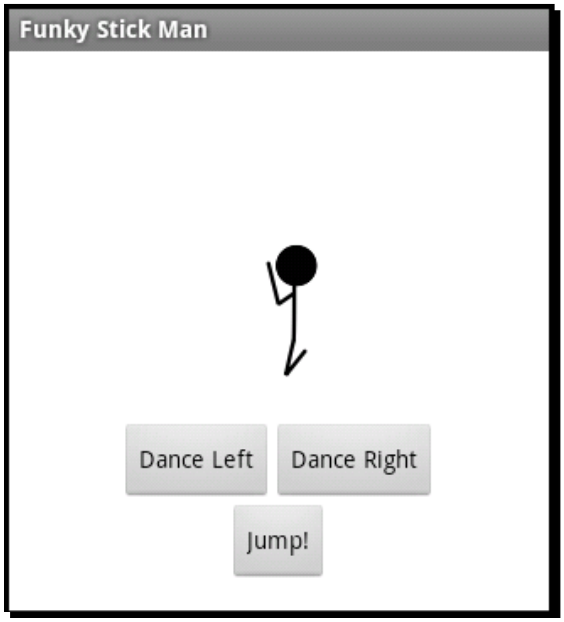
6. At last! Time to wire up the button to do the jumping. At the end of the `onCreate()` method, add the following:

```
Button jumpButton = (Button) findViewById(R.id.jump);
jumpButton.setOnClickListener(new View.OnClickListener() {
  public void onClick(View v) {
    animImage.setImageDrawable(jumpAnim);
    jumpAnim.start();
    jumpAnim.setVisible(true,true);
  }
});
```

It looks pretty similar to the `Button onClickListeners` that you made for the other dance moves, doesn't it? There are two main differences though! Firstly, you don't need to retrieve the `AnimationDrawable` by doing `getResources().getDrawable(int id);` anymore, as you can access the `jumpAnim` stored locally in the `FunkyActivity`.

The second difference is subtler, and it will drive you up the wall if you forget it! Because you are re-using the same animation whenever you press the button, it remembers that it has already been played. Unless you reset the animation back to its start position, it will do nothing and just stay stuck on its last frame. You use `jumpAnim.setVisible(true,true);` to reset the animation position.

7. We're finished! Let's test the new move.



What just happened?

Here we built a completely new animation from a set of Drawables. We wrote a method that programmatically queried an `AnimationDrawable`, and we used it to programmatically create another `AnimationDrawable`.

More neat methods on AnimationDrawable

In the previous *Time for action – programmatically defined animation* section, we made use of some additional methods that you can use to build on your animation. These methods are more geared to programmatic frame animation.

setOneShot(boolean oneShot)

This is the programmatic equivalent of the `android:oneshot` XML attribute. If you set it, your animation will stop when it reaches its last frame.

addFrame(Drawable frame, int duration)

Adds a frame to the end of your animation and shows it for the specified time. This is the key method to use when building up an animation in a programmatic way.

getFrame(int index)

This is useful when finding out information about an animation, for instance, when you want to create a new animation based on some transformation of the old animation.

getDuration (int index)

The sister of `getFrame()`. If you want to read off the durations of your animation frames, this is the thing to use.

getNumberOfFrames()

When you are doing programmatic work on an animation and you're not absolutely certain how large it is, you can call `getNumberOfFrames`. This is particularly useful when using `getFrame` to avoid trying to get frames that don't exist.

Working properly in the GUI thread

When you are manipulating animations in Java, always be aware of the thread from which your control code will be called—a great deal of operations on views and Drawables can only take place when you are running in a special thread. This thread is commonly known as the GUI thread. It is just like any other thread in Android, but the following are a couple of instances where you must be careful when using it:

- When an activity starts, it goes through the `onCreate` method to determine what it is actually going to show. By calling `setContentView`, you load the views that you want to use into memory. But they are not necessarily ready to use. They are like an unfinished kit, and they need to be measured and laid out onto the display. Android processes the display after the `onCreate` method has returned, making it ready to start displaying on the screen.

  Although `onCreate` runs in the GUI thread, if you try modifying a view in `onCreate`, it may or may not work, depending on the thing that you are trying to change. Animations many not get applied and dimension measurements may be wrong because the views have not been positioned on screen yet.

- Once your activity has started running, all changes to the GUI must be made from the GUI thread. The GUI thread maintains strict ownership of all of the views that it draws.

  If you try to modify a view from outside of the GUI thread, you will receive a stern exception and your application may force-close.

So we have two cases where we must be careful how we call methods on views that might modify them. Fortunately, there is a handy method on all Android views that makes it easy to use the right thread at the right time. If you create a `Runnable`, and you pass it to `View.post(Runnable r)`, Android will run it on the GUI thread as soon as the GUI thread is ready to execute it. This is a great way to guarantee that your GUI code is run at the right time in the right thread.

In the previous example, you will see that we called `startAnimation` from within the `onCreate` method by making a post call. This ensures that our animation will be added to the view only once the view is ready to receive it.

Android GUIs are multi-threaded, and Android has a special thread that is dedicated to running GUI code. If you intend to make changes to GUI elements in Java, you should use `View.post(Runnable r)` to run your GUI code in the GUI thread, unless you are sure that it will be run in the GUI thread anyway (for instance, if it is only called as part of an event handler such as `View.OnClickListener()`).

Pop quiz – controlling frame animations

1. What is the Java class that represents an Android frame animation?
   a. `FrameAnimation`
   b. `AnimationDrawable`
   c. `DrawableFrame`
2. What is the secret function of `AnimationDrawable.setVisible(true,true)`?
   a. It allows you to edit an uneditable Animation
   b. It transports you to Narnia
   c. It resets the `AnimationDrawable` to its first frame
3. When should you call `View.post(Runnable r)`?
   a. When you want something to run later
   b. When you are running `findViewById(int id)`
   c. When you are calling `display` methods on Views and Drawables.

Have a go hero – reactive frame animation

When the `Funky Stick Man` loads, he is full of energy! However, every time you click a button, he gets just a little more tired and takes a little longer to do his dance. Update the `Funky Stick Man`, so that whenever you press a button each frame is shown for an extra `3` milliseconds.

If you get it right, he will get a little jerky after a few clicks. Don't worry! That's fine.

Here are some clues to help you along:

- You will need to remake the dance animations every time you change the durations. Some sort of copying routine will be useful to help you remake the animations.
- You will need to store the new duration somewhere, and increase it every time someone hits a button.

Animating a transition between frames

After the myriad frames in frame animations, you'll be pleased to know that the next sort of animation, the **transition** animation, is much simpler. We want an animation to fade smoothly from one image to another, so that it is not just a sudden swap.

The translation is simply a fade between two frames. It can be used for showing that something has been selected or deselected, or for gradually introducing a new visual element to a scene.

Time for action – make the transition

This time, we are going to make an activity that reveals a hidden scene when you touch it. The scene is going to be made from two images. The first image will be a **lights off** image, where you can only see a vague shape, as if it was very dark. The second image will be the scene itself, with a light bulb in the scene. We will use a transition animation to give the impression that the light is lit slowly, by fading between the first and second images.

1. Create a new Android project with the following settings:
   - **Project name:** `NightLight`
   - **Build Target:** `Android 3.0`
   - **Application name:** `NightLight`
   - **Package name:** `com.packt.animation.nightlight`
   - **Create Activity:** `NightLightActivity`

2. Firstly we need to import the graphics `lightoff.png` and `lighton.png` from the code bundle. Unzip `tutorial_images_2` to a directory on your hard disk and copy the files inside to your project under `NightLight/res/drawable`. These images become our start and end scene images.

3. Without further ado, let's make the transition animation! We want to create a fade from dark to light. Create a new XML file in `res/anim` called `lighton.xml`, and put the following in it:

```
<?xml version="1.0" encoding="utf-8"?>
<transition
  xmlns:android="http://schemas.android.com/apk/res/android">
  <item android:drawable="@drawable/lightoff" />
  <item android:drawable="@drawable/lighton" />
</transition>
```

   As you can see, we add the images in a very similar way to how we added the frames to the frame animation. And it will become a Drawable object in Java, so the two types of animation are closely related in this sense. But you have probably noticed that there are no `android:duration` elements this time; we will define the time to be taken for fading, a bit later on.

4. Next, we need to tell the application to include the image in the main layout of the application. Open up `res/layout/main.xml` and change its contents to the following:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/mainscreen"
  android:orientation="vertical"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  android:gravity="center"
  android:background="#FF000000">
  <ImageView
    android:id="@+id/lightgraphic"
```

```
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:src="@anim/lighton"
  />
</LinearLayout>
```

As you can see, we are just treating the animation as the body of an `ImageView`, as we have done previously. We have given it the ID `lightgraphic` to use later in Java code.

Notice that we have also given the layout an `android:id` this time. This is just so that we can use it as a **touch** button in Java (so that the user can touch anywhere on the screen to activate the animation).

5. If we just include the animation like this, it will be shown but it won't start playing back. That is, if we don't add any Java code, then it would stay as a still image. It would look as if we had written, `android:src="@drawable/lightoff"`.

In order to convince you of this, let's build and run the `NightLightActivity` now.

No matter what you do, the activity stays mysteriously dark, doesn't it?

6. Now to add the code. We don't want the light to come on straight away, so we will wait until the user has touched the screen. To achieve this, we're going to put it in an `OnClickListener`, but this time we will set it to detect any touches to the whole application.

Firstly, we'll add a couple of new classes to the `NightLightActivity`:

```
import android.view.View;
import android.widget.ImageView;
```

These are just here to help wire in the controls to the new animation.

```
import android.graphics.drawable.TransitionDrawable;
```

And this is the subject of this tutorial, the `TransitionDrawable` itself!

7. Next, in the `NightLightActivity`, you want to define your `onCreate` method, as shown in the following code. The button stretches across the whole screen, so you can touch anywhere to start the animation.

```
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.main);
  View mainScreen = findViewById(R.id.mainscreen);
  mainScreen.setOnClickListener(new View.OnClickListener() {
    public void onClick(View v) {
      // We're going to fill this in in a moment
    }
```

```
  });
}
```

Once again, we use the code from the default activity to set the `main.xml` as the content view. Then we locate the top-level visual component (`R.id.mainscreen`) and assign it a `View.OnClickListener()`.

8. Now, let's fill out the `onClick` event.

```
ImageView scene =
  (ImageView) findViewById(R.id.lightgraphic);
TransitionDrawable sceneDrawable =
  (TransitionDrawable) scene.getDrawable();
  sceneDrawable.startTransition(3000);
```

This locates the actual `TransitionDrawable (R.id.lightgraphic)`, much as we have done for the other examples.

Then it calls `startTransition(int duration)` on the Drawable. Here is where the duration is defined!

The figure of `3000` is in milliseconds, which is enough to give us a nice satisfying **warming up** transition.

9. That's it! Build, deploy, and enjoy!

What just happened?

Here we took a look at an animation that is defined in a very similar way to a frame animation, but where the animation itself is created programmatically. A fade might not seem very exciting compared to the complex detail of a frame animation, but it can be used to make your user interfaces look much smoother.

Writing XML for a transitionDrawable

The transition XML element is a little bit like the frame animation's animation list, but there are a few differences. For reference, I've described them here.

As before, the main headings are XML tags, and the subheadings are XML elements within those tags.

<transition>

`animation-list` is always the top-level element in a frame animation. Again, the animation list is an ordered container of Drawable items, but there are only two frames.

The first frame in the list becomes the start frame and the second frame in the list becomes the end frame. Don't put any more `items` in the frame; Android won't know what to do!

```
<transition
  xmlns:android="http://schemas.android.com/apk/res/android">
```

This time around, we get rid of the `oneShot` attribute. Transition animations are not really suited to looping.

This tag should contain the following options:

xmlns:android

All top-level Android XML elements declare their namespace as `xmlns:android="http://schemas.android.com/apk/res/android"`

<item>

Each one of the frames in the list is described as an `item` that references a pre-compiled Drawable.

```
<item android:drawable="@drawable/lighton" />
```

Each `item` has only one attribute, namely, a Drawable.

android:drawable

As with the frame animation, this is a required attribute that specifies the start or end frame of the animation.

Working with other useful methods

`TransitionDrawables` have about as many controls as frame animations, that is, not many! Note that the `TransitionDrawables` only have two steady states, and they don't need you to call `setVisible(true,true)` in order for them to work repeatedly.

startTransition(int duration)

As you might think, this kicks-off your animation, and fades from the first frame to the second. The duration is in milliseconds.

reverseTransition(int duration)

One control that you might find particularly handy is the `reverseTransition(int duration)`. As you might expect, it is the exact opposite of `startTransition(int duration)` and takes your animation right back to its start frame. This is very handy for writing animations that toggle between two states.

resetTransition()

Of course, maybe you just want to get back to the start frame, without displaying any animation. Just call `resetTransition` and you're there.

Pop quiz – transition drawables

1. A transition is a kind of
    a. View
    b. Widget
    c. Drawable
2. How do you reverse a transition?
    a. Call `reverseTransition`
    b. Take a copy of the transition frames and make a new transition
    c. Call `startTransition` with a negative argument

3. How many frames does a `TransitionDrawable` have?
   a. 0
   b. 1
   c. 2
   d. As many as you like
4. When you define an XML transition, you need to include:
   a. `android:duration` values
   b. A start and end frame
   c. A `oneShot` declaration

Have a go hero – transition Drawables

When you touch the screen in the `NightLightActivity`, the transition goes from light to dark and the light comes on. But what happens when you touch it again? The light suddenly goes off and comes back on again! That's not very realistic, and it would be cool if the second time you touch the screen, the light fades off again.

If only there was a way to play the transition in reverse…

Your task is to make the `NightLight` application do the following:

- When the light is off and the user touches the screen, fade the light on
- When the light is on and the user touches the screen, fade the light off
- You should be able to do this repeatedly: light on, light off, light on, light off…

Think you know how to do it? Away you go!

Summary

We learned a lot in this chapter about frame and transition animations; two ways with which you can add animation to an application using frames.

Specifically, we covered:

- Defining an animation in XML, using a series of frame images
- Using controls on animations
- Making copies of an animation in order to modify them
- How to define and use a `TransitionDrawable` in an activity

Although they are quite similar to implement, the visual effects of frame and transition animations can be quite different.

We also discussed portability, and how you can use Android features to make your animation look good on different sized screens. We saw that we must take care not to make our animations too big to run on a small Android device. We also noted that you must be careful to run your graphics code in the GUI thread.

You now know everything there is to know about frame animations! In the next chapter, we'll take a look at tweening, a different class of animation that adds animated visual cues to boring old form elements.

3
Tweening and Using Animators

In the previous chapter, we saw how to take an existing animation and add it to an Android layout. In this chapter, we shall look at how to take ordinary views and widgets within a layout and add animation to them.

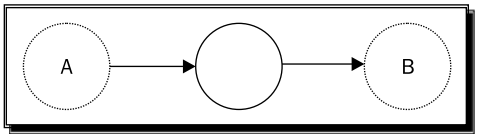
The main points we shall cover include:

- Defining a tween in XML
- The tweening operations that are provided with Android
- Defining and parameterizing an Animator

Let's start by making our first tween.

Greeting the tween

Tweens, you will recall, are animations that move between two states. They are atomic elements in Android, and exist completely independent of individual views. In this way, you can apply the same tween to lots of different things to give your application a consistent look and feel in its animation style.

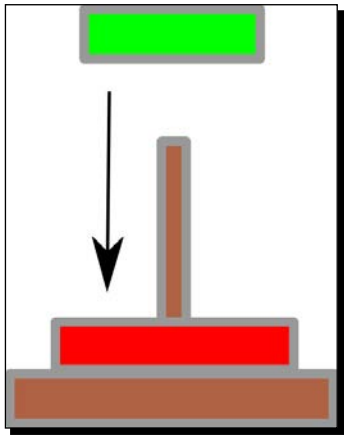


When you use a tween in an activity, you usually want to define it in advance, so that you can apply it consistently whenever you need to move, or shake, a view from one place to another. As with other graphical elements, Android provides an XML format for describing tweens as a collection of component parts.

Time for action – making a tower of Hanoi puzzle

The **Towers of Hanoi** is a classic logical puzzle that you have almost certainly played, maybe even written a computer program to solve it. It involves stacking several pieces in height order onto one of three pegs.

In this chapter, we are going to create that game in a simple sort of way. Don't worry if you have never played it before; I'll explain before each tutorial what we are going to make next.



Here we see a side-view of a tower being built. The peg is shown in brown, the red block is resting on the peg, and the green block is being added to the peg. The height of the tower is the number of blocks on its peg.

In this tutorial, we will make a peg graphic onto which we can put building blocks. The blocks will arrive on the peg (actually, just a `LinearLayout`) using a tween animation. This is the way a tower is formed in the game.

1. Create a new Android project with the following settings:
   - **Project name:** `Towers of Hanoi`
   - **Build Target:** `Android 3.0`
   - **Application name:** `Towers of Hanoi`
   - **Package name:** `com.packt.animation.hanoi`
   - **Create Activity:** `HanoiActivity`

2. We want building blocks to fall from the sky to build up the tower piece-by-piece. Let's introduce a tween that moves a new building block from the sky to the peg.

First, create a new Android XML file in `res/anim` and call it `block_drop.xml`. In it, add the following XML root element:

```
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android="http://schemas.android.com/apk/res/android"
  android:interpolator="@android:anim/linear_interpolator">
</set>
```

Here, we create a top-level container for any number of tween animations. The `android:interpolator` describes the rhythm of the animation.

There are a selection of pre-defined Android interpolators, and `@android:anim/linear_interpolator` means *move from the start point to the end point at a constant speed*. Later on, we will see how to use a few different interpolators in an animation.

3. For now, we just want to translate a block from one place to another. Inside the `<set>` tags, add the following code:

```
<translate
  android:fromYDelta="-100%p"
  android:toYDelta="0"
  android:duration="3000"/>
```

There are a couple of things going on here, so let's briefly explain them.

- The `android:fromYDelta` and `android:toYDelta` are the start and end positions for the animation. We are going to make something fall from the sky, so we are only interested in the `Y` coordinate; `X` will stay constant.
- The `duration` is the length of time that the `<translate>` should take to complete, in milliseconds.

4. Now we need to build our scene. Our tower will be a simple `LinearLayout`, and the blocks will be `TextViews`. In `res/layout/main.xml`, set your layout to look like the following:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="horizontal"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  android:gravity="bottom|center_horizontal">
  <LinearLayout
    android:layout_width="fill_parent"
```

```
    android:layout_height="fill_parent"
    android:orientation="vertical"
    android:gravity="bottom|center_horizontal">
    <TextView
      android:id="@+id/block_1"
      android:layout_width="70sp"
      android:background="#FF448844"
     android:layout_height="wrap_content"
      android:gravity="center"
      android:text="1"
    />
  </LinearLayout>
```

</LinearLayout>

For now, we will just add one block; it's labeled `block_1` in the XML above. Its color will be used to distinguish it from other blocks, which we will add soon.

5. Now we have a scene with a block in it, but it's not animated yet! We use Java to apply the animation to the block. Open up `com.packt.animation.hanoi.HanoiActivity` and add the following headers:

```
import android.view.animation.Animation;
import android.view.animation.AnimationUtils;
import android.widget.TextView;
import android.view.ViewGroup;
import android.view.View;
```

The `Animation` interface is the Java-based interface of all tween animations; we use it for passing the animation to the `TextView` `block_1`.

The `AnimationUtils` provides a mechanism for creating an instance animation from the definition given in the `res/anim` directory.

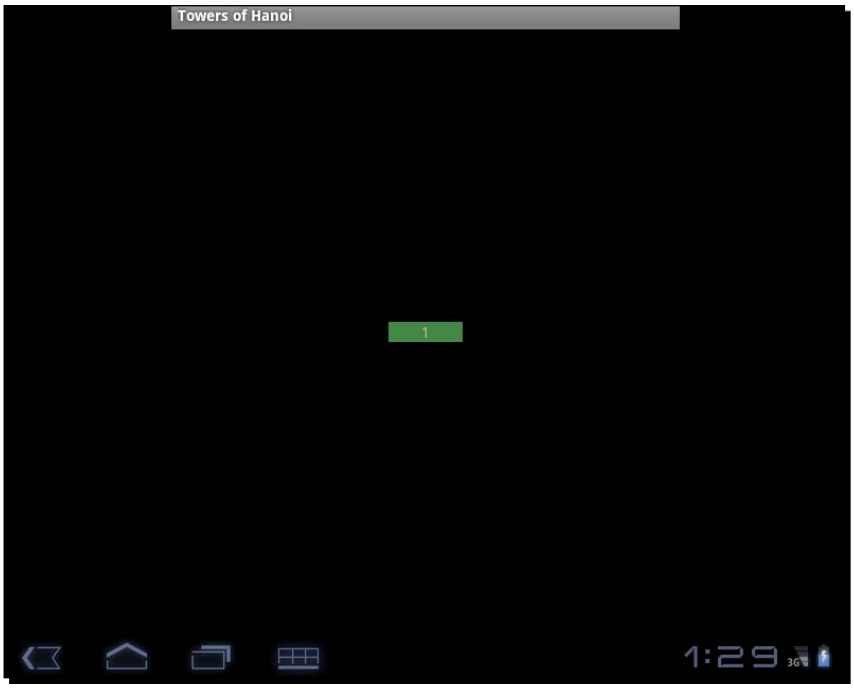
6. Now, at the end of the `onCreate()` method, add the following lines of Java after the `setContentView` line:

```
TextView block1 = (TextView) findViewById(R.id.block_1);
Animation drop = AnimationUtils.loadAnimation(
  this, R.anim.block_drop);
```

block1.startAnimation(drop);

This picks out our block from the scene and loads our animation from the compiled resource. Then, we simply call `startAnimation` to start applying the animation to the block.

7. Okay, now we have something we can watch! Build and run the application - you should see a block fall from the sky and then stop. That is the basic process through which we will place all blocks onto the tower.



What just happened?

You just made a tween animation and applied it to an ordinary view in a layout.

The tween was defined as an XML file, and loaded at runtime by the `AnimationUtils` class.

Defining starts and ends

The tween animation has a notion of its start state and its end state. In a `<translate>` tween, the start and end states correspond to the start and end positions on screen.

If you look at the animation defined in `block_drop.xml`, we used the following parameters to position the animation. Notice that they are generic and that they do not depend on a particular view or layout.

<translate> Attribute	The Value We Used	What Does That Mean?
`android:fromYDelta`	`-100%p`	Start the animation from the top-most part of the parent [`p`] object.
`android:toYDelta`	`0`	End the animation at the view's resting position, namely the position it would be if we weren't doing an animation.
`android:fromXDelta`	`Omitted`	This is the same as setting them to `0`. We don't want to move the block along the `X`-axis at the moment.
`android:toXDelta`		

You will see later that other tweens have different notions of starts and ends, but they all represent the idea of getting from one visual point to another.

Assembling the building blocks of a tween

There are different sorts of tweens, and they can be combined in series or in parallel to make quite complex animations. The different types are:

- `Translate`: Move from one place to another
- `Rotate`: Spin around
- `Scale`: Get bigger or smaller
- `Alpha`: Fade in or out
- `Set`: Yes, a set is a tween too, and you can nest them! `Set` lets you define certain properties for all of the tweens contained within it.

Let's explore this idea with an example.

Time for action – composing a tween animation

In the full Towers of Hanoi game, we don't just put the blocks in one place, but we move them about between different towers. Let's make an animation to describe flipping the block from one place to another.

1. First up, we need to add a second and third tower, so that our block has somewhere to be moved to. In `main.xml`, update it, so that it looks like the following:

```
<?xml version="1.0" encoding="utf-8"?>
```

```
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="horizontal"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  android:gravity="bottom|center_horizontal">
  <LinearLayout
 android:id="@+id/tower_1"
    android:layout_width="200sp"
 android:background="#FFEEDDDD"
    android:layout_height="fill_parent"
    android:orientation="vertical"
    android:gravity="bottom|center_horizontal">
    <TextView
      android:id="@+id/block_1"
      android:layout_width="70sp"
      android:background="#FF448844"
      android:layout_height="wrap_content"
      android:gravity="center"
      android:text="1"
    />
  </LinearLayout>
  <LinearLayout
 android:id="@+id/tower_2"
    android:layout_width="200sp"
 android:background="#FFDDEEDD"
    android:layout_height="fill_parent"
    android:orientation="vertical"
    android:gravity="bottom|center_horizontal"
  />
  <LinearLayout
 android:id="@+id/tower_3"
    android:layout_width="200sp"
 android:background="#FFDDDDEE"
    android:layout_height="fill_parent"
    android:orientation="vertical"
    android:gravity="bottom|center_horizontal"
  />
</LinearLayout>
```

Things to note are that we are adding identifiers for the towers, which will make them easy to tell apart. Visually, we've also given them some background color. We have left `block_1` residing in `tower_1`.

2. Next up, we are going to create a new animation. In `res/anim`, create a new file called `block_move_right.xml`. This will lift a block up from its tower, and make it fly off upwards and to the right. We will also make it spin around as it moves.

3. Before we add a new animation, let us consider what we actually want it to do.
   - Lift the block up off of its tower
   - Lift it up and to the right
   - Make it spin

   In `block_move_right.xml`, add the following XML. I will interrupt at various points to explain it:

```
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android="http://schemas.android.com/apk/res/android"
  android:interpolator="@android:anim/linear_interpolator">
  <rotate
      android:fromDegrees="0"
      android:toDegrees="360"
      android:pivotX="50%"
      android:pivotY="50%"
      android:duration="1500"
      android:startOffset="1500" />
```

   This is the `rotate` that I mentioned. Notice that it has different settings from the ones on the translate operations. I'll talk about them in a minute, but you can probably guess what they do.

   The other new attribute is `startOffset`. In a compound animation like this one, you might not want all of your animation elements to begin at the same time. Here is where `startOffset` comes in; this animation won't start playing until the animation, as a whole, has been running for `1500` milliseconds.

```
  <translate
    android:toYDelta="-100%p"
    android:fromYDelta="0"
    android:duration="3000"
  />
```

   This is the upward motion of the animation.

```
  <translate
    android:toXDelta="100%"
    android:startOffset="1500"
    android:duration="1500"
  />
```

And here is the rightwards part of the motion. Like the `spin`, we wait until `1500` milliseconds have passed, before we move the block to the right. This is because we want to show the block being lifted up before it is moved to one side.

</set>

4. Now to add it to `HanoiActivity.java`. To make it easy to navigate the different towers, you should add a static reference to them at the top of the `HanoiActivity` class as follows:

```
private static int[] towers = {
  R.id.tower_1,
  R.id.tower_2,
  R.id.tower_3
};
```

Now we can refer to the towers by index instead of by a hardcoded ID. This will be useful later on.

5. We will create a new class to look after our block moving animations. Inside the `HanoiActivity` class, create an inner class called `BlockMover` as follows:

```
private class BlockMover {
  private int to, from;
  View block;
  protected BlockMover (View block, int from, int to) {
    this.block = block;
    this.to = to;
    this.from = from;
  }
  public void move() {
    // We will fill this in in a minute
  }
}
```

We will store indexes to towers in the `to` and `from` fields, and this block will be the one that we want to animate. Let's fill in the green bit right now.

6. Add the following code inside the `move()` method:

```
public void move() {
  Animation removeAnimation = AnimationUtils.loadAnimation(
    HanoiActivity.this, R.anim.block_move_right);
  block.startAnimation(removeAnimation);
}
```

This is very like the way we added the first animation.

7. We will also want some way of launching this animation, and choosing which tower to move it to. We will add another inner class to do this, inside `HanoiActivity` but outside `BlockMover`. This one will be a `View.OnClickListener`, so that it can take actions when a tower gets clicked on. Create the following class:

```
private class TowerPicker implements View.OnClickListener {
  private int towerIndex;
  public TowerPicker (int towerIndex) {
    this.towerIndex = towerIndex;
  }
  public void onClick(View v) {
    // We will fill this in in a minute
  }
}
```

8. When a user clicks on a tower, he/she is selecting it. When he/she then clicks on another tower, he/she is saying *move a block from the first tower to the second*. This means that we should record the first tap, so that the `TowerPicker` knows which tower it is in. We will do this by storing a member variable in `HanoiActivity`. Outside our new inner class, add in the following lines:

```
private static final int UNDECIDED = -1;
```

private int fromTower = UNDECIDED;

Now when there are no towers selected, the `fromTower` should be `UNDECIDED`. And when there is already a selected tower, it will be the index of the tower in `towers`.

9. Now to add in a bit of game logic that decides when we need to run a `BlockMover`. There is an important game rule here; I'll interrupt briefly as that game rule is introduced. The following code is to be added to `TowerPicker.onClick` just after // *We will fill this in in a minute*

```
if (fromTower == UNDECIDED) {
  ViewGroup tower =
    (ViewGroup) findViewById(towers[towerIndex]);
  if (tower.getChildCount()>0) {
    fromTower = towerIndex;
  }
} else {
  ViewGroup fromTowerView =
    (ViewGroup) findViewById(towers[fromTower]);
  if (fromTower != towerIndex) {
    ViewGroup toTowerView =
      (ViewGroup) findViewById(towers[towerIndex]);
    View block = fromTowerView.getChildAt(0);
```

View supportingBlock = toTowerView.getChildAt(0);

Only add a block to a tower if that tower is empty, or if its top block is bigger than the block you are adding.

```
    if (supportingBlock == null
      || supportingBlock.getWidth()>block.getWidth()) {
        (new BlockMover (block,fromTower, towerIndex)).move();
      }
    }
    fromTower = UNDECIDED;
}
```

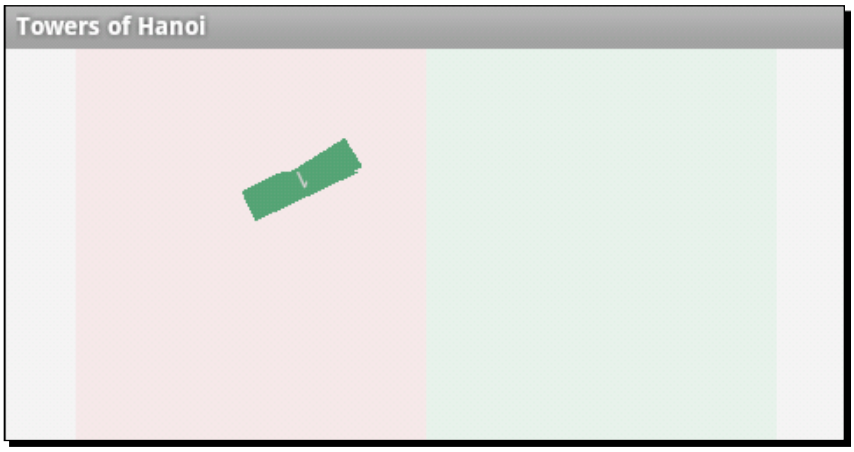
This is our structure for initiating the block move animations.

10. Finally, at the bottom of the `onCreate()` method, add the following line:

```
for (int i = 0; i < towers.length; ++i) {
  ViewGroup tower = (ViewGroup) findViewById(towers[i]);
  tower.setOnClickListener(new TowerPicker(i));
}
```

When a user clicks on a tower, a `TowerPicker` will handle the clicks, and a `BlockMover` will be used for animating it to the next tower.

11. Now when you build and run the activity, you will be able to fling the block over to the left, simply by tapping on it.



What just happened?

By putting multiple elements into `<set>` tags, we built up a complex animation. The `rotate` animation is parameterized in a different way to the `translate` animation, but they still have common elements like the `duration` and `startOffset` attributes. We used the `startOffset` attribute to do multiple animations in a sequence.

You will have noticed that the block doesn't actually take advantage of any of these rules yet, as it's stuck in one tower. Don't worry, we will make that part of the game work when we get to animating events, and we learn how to do things at the end of our animation.

We also introduced you to the rules of this game; there weren't many!

Taking a look at the different types of tween animation

There are several tween animation types that each do different things. I will list them here for reference; you can combine them in many ways to get different effects.

<translate>

As you have seen, this moves a view from one place to another.

Translate Attribute	Values	Meaning

`android:fromXDelta`
`android:fromYDelta`
`android:toXDelta`
`android:toYDelta`
	A percent relative to itself `"75%"`, a percent relative to its parent `"-50%p"`, or an absolute number of pixels `"43.5"`	`from?Delta` refers to the start point on that axis
		`to?Delta` refers to the end point

<rotate>

It makes an element spin about an axis. We used this one in the previous tutorial too.

Rotate Attribute	Values	Meaning
`android:fromDegrees`	A rotation value in degrees	`fromDegrees` refers to the rotation start point
`android:toDegrees`		`toDegrees` refers to the end point

Rotate Attribute	Values	Meaning

`android:pivotX`
 	
`android:pivotY`
	A percent relative to itself `"75%"`, a percent relative to its parent `"-50%p"`, or an absolute number of pixels `"43.5"`	The pivot point is the place from which rotation takes place

<alpha>

It changes the alpha level of a view to make it fade in or out.

Alpha Attribute	Values	Meaning

`android:fromAlpha`
 	
`android:toAlpha`
	An alpha value from `0` (transparent) to `1` (solid)	`fromAlpha` refers to how solid the view is when the animation begins. `toAlpha` is the solidity when it ends

<scale>

It makes the view bigger or smaller.

Scale Attribute	Values	Meaning

`android:fromXScale`
 	
`android:fromYScale`
 	
`android:toXScale`
 	
`android:toYScale`
	A scale multiplier where `1.0 = normal size`	`from?Scale` refers to the starting scale
		
		`to?Scale` refers to the end scale

`android:pivotX`
 	
`android:pivotY`
	A percent relative to itself `"75%"`, a percent relative to its parent `"-50%p"`, or an absolute number of pixels `"43.5"`	The pivot point is the place from which scaling takes place

Common attributes

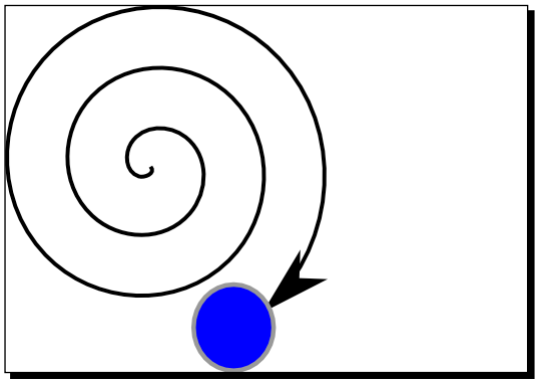
These attributes can be used with all tweens.

Attribute	Values	Meaning
`android:interpolator`	An Android `@reference` to an interpolator	The interpolator provides the rhythm of the animation. We will take a closer look at this later on in this chapter.
`android:duration`	Millisecond value	How long to make the animation last for.
`android:startOffset`	Millisecond value	How long to wait before starting this section of the animation.
`android:repeatCount`	Integer repeats/`"infinite"`	How many times to repeat this animation. Note that the overall duration of the animation will become `duration` x `repeatCount`.
`android:repeatMode`	`"restart"/"reverse"`	What to do when repeating this animation. You can either leap back to the start position and repeat from there, or hit the rewind button and go back to the start smoothly.

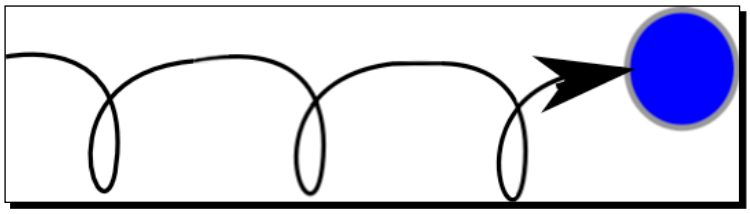
Declaring tweens in the correct order

The order in which you declare tween elements can be important.

If in your XML, you declare a translate **tween** and then a **rotate** tween; the rotation is going to happen relative to the place where the animation started, making a spiral motion.



If you declare the rotate element before the translate element, the view will rotate relative to its moving position, making a corkscrew motion.



In the previous example, we wanted the view to rotate around its line of motion, so we chose the second option.

Making tweens that last for ever

From what we've talked about, you might think that tweens have to stop on their own. They have a start and an end point, right? Ah! But look up to the section entitled *Common Attributes*, and you'll see that you can specify an `android:repeatCount` of `infinite`. Let's have a look at how this might be useful.

Time for action – creating an everlasting tween

At the moment, the user has no visual clue as to whether he/she has clicked a start block yet. Perhaps he/she got distracted and can't remember which block he/she touched last.

Let's add a new animation which makes the **from** tower glow gently, that is, the tower from which the block will be taken. When our user selects a **destination** tower, the `move` operation will be complete and we can stop the pulsing animation.

1. Create a new XML file in `res/anim` called `tower_glow.xml`. This is going to be the animation that we apply to the selected tower.

2. In this new file, add the following XML:

```
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android="http://schemas.android.com/apk/res/android">
  <alpha
    android:fromAlpha="1"
    android:toAlpha="0.7"
    android:repeatCount="infinite"
    android:repeatMode="reverse"
    android:duration="1000"
  />
</set>
```

Here we are using an `<alpha>` animation, as described above. We are also using the `android:repeatCount="infinite"` trick that I mentioned. This animation will last until we manually delete it. Also, notice that the `android:repeatMode` is set to `"reverse"`, which means that, when the fade is at `0.7`, it will smoothly transition back to `1` rather than snapping back immediately. Take a good look and imagine how it's going to appear to the user.

Underneath the tower, the background will be black.

3. Now we need to start the animation when the user clicks on their first tower, and we need to stop it when they click on a second tower.

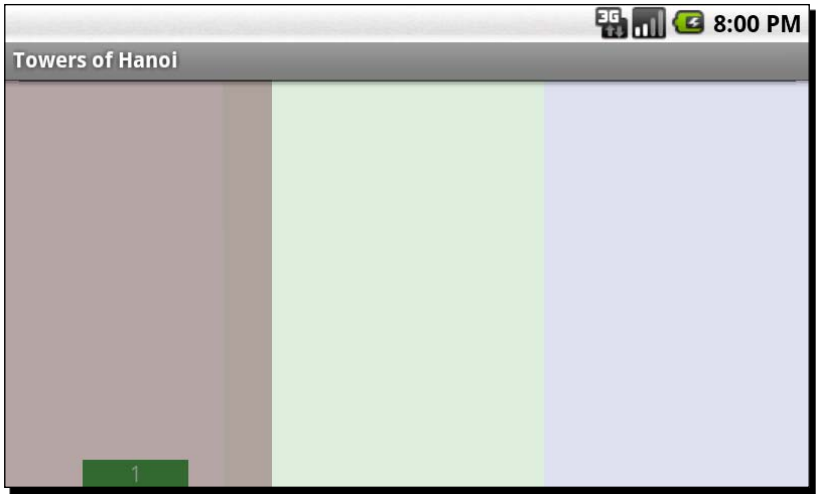
Open up `HanoiActivity.java` and navigate to the `onClick` method in `TowerPicker`. Add in the following lines I've highlighted:

```
public void onClick(View v) {
  if (fromTower == UNDECIDED) {
    ViewGroup tower = (ViewGroup)
      findViewById(towers[towerIndex]);
    if (tower.getChildCount()>0) {
      fromTower = towerIndex;
   Animation glowAnimation =
     AnimationUtils.loadAnimation(
       HanoiActivity.this,
       R.anim.tower_glow
   );
   tower.startAnimation(glowAnimation);
    }
  } else {
  ViewGroup fromTowerView =
    (ViewGroup) findViewById(towers[fromTower]);
    if (fromTower != towerIndex) {
      ViewGroup toTowerView = (ViewGroup)
        findViewById(towers[towerIndex]);
      View block = fromTowerView.getChildAt(0);
      View supportingBlock = toTowerView.getChildAt(0);
      if (supportingBlock == null
        || supportingBlock.getWidth()>block.getWidth()) {
        (new BlockMover(
          block,fromTower, towerIndex)
        ).move();
      }
```

```
        }
    fromTowerView.clearAnimation();
        fromTower = UNDECIDED;
      }
}
```

This should look pretty familiar by now. The only new thing is that we are calling `clearAnimation()` to return the tower back to its deselected, non-glowing state.

4. Build and run it, and see how it looks!



What just happened?

Now, when the user touches a well, the top brick in it flashes indefinitely; we have made an animation that loops forever. It is still a tween animation, because it goes between two different states: **translucent** and **solid**. But the behavior of an infinite animation is different, and we had to stop it from within Java code.

To get the animation to loop indefinitely, we used the attribute `android:repeatCount="infinite"` from within the XML animation description. We also characterized the animation to reverse its animation when it finished, rather than jumping immediately back to its start state. For this, we used the attribute `android:repeatMode="reverse"`.

In order to stop the animation, we had to call `View.clearAnimation()` from an event handler in Java. Otherwise, the animation would loop forever!

Pop quiz – all those tweens !

1. If you want to animate something fading away, what tween would you use?
   a. Rotate
   b. Translate
   c. Alpha
   d. Scale
2. What should you be aware of when declaring translate and rotate animations together?
   a. It isn't important
   b. Whether the centre of rotation will be translated or remain fixed
   c. Whether they have different interpolators
3. If you want to animate something getting bigger, what tween would you use?
   a. Rotate
   b. Translate
   c. Alpha
   d. Scale
4. How do you combine different tweens together in one animation?
   a. Use a Java class to order them
   b. Use a `<set>` element
   c. Declare one tween inside another
5. How would you put two different tweens in sequence in the same animation?
   a. Use the `duration` and `startOffset` to control their order
   b. You can't
   c. Declare the tweens in the order you want to run them

Have a go hero – bouncing back

As you may have noticed, we can now send the block to the left or to the right. But we only have an animation for going to the right! Create a new `block_move_left` animation like the `block_move_right` animation, but going the other way.

To get you started, I'll give you a piece of code that you might want to use. In the `BlockMover` `onClick()` method, replace this piece of code:

```
Animation removeAnimation =
  AnimationUtils.loadAnimation(
    HanoiActivity.this, R.anim.block_move_right);
```

With this new piece of code:

```
int block_anim_id;
if (to<from) {
  block_anim_id = R.anim.block_move_left;
} else {
  block_anim_id = R.anim.block_move_right;
}
Animation removeAnimation =
  AnimationUtils.loadAnimation(
    HanoiActivity.this, block_anim_id);
```

If you're feeling adventurous, try creating your own style of block movement.

Animating layouts

You can trigger an animation to apply, as soon as a container is laid out. This gives the feeling that the scene is being assembled when you first see it.

Time for action – laying out blocks

We are going to add some more blocks to our tower, to make the game more playable. Rather than writing a load of Java calls to add the new blocks, we can use a layout animation to add them all into the scene.

1. Create a new XML file in `res/anim` called `layout_tower.xml`. This will be the layout animation for `tower_1` when the game starts.

2. In this new file, add the following XML:

```
<?xml version="1.0" encoding="utf-8"?>
<layoutAnimation
  xmlns:android="http://schemas.android.com/apk/res/android"
    android:animation="@anim/block_drop"
    android:delay="20%"
    android:animationOrder="reverse" />
```

This is all you need to declare your layout animation!

- The actual `android:animation` itself is just a reference to the `block_drop` animation that we defined earlier.
- The `android:delay` is a small gap between the animations of each block that we want to add.
- The `android:animationOrder` is the order in which the layout animation adds each element. Ordinarily, it would start with the first element, which in a vertical `LinearLayout` would be at the top. We want it to start with the last element, because we put the blocks on top of each other.

3. Next, let's apply the new animation to our layout. In `res/layout/main.xml`, add the following line to the `LinearLayout` called `tower_1`:

android:layoutAnimation="@anim/layout_tower"

Of course, this won't look too different from before, because we are applying the same layout to the same blocks. Let's make it a bit more useful by adding in some more blocks underneath `block_1`.

```
<TextView
  android:id="@+id/block_1"
  android:layout_width="70sp"
  android:background="#FF448844"
  android:layout_height="wrap_content"
  android:gravity="center"
  android:text="1"
/>
<TextView
  android:id="@+id/block_2"
  android:layout_width="100sp"
  android:background="#FF444488"
  android:layout_height="wrap_content"
  android:gravity="center"
  android:text="2"/>
<TextView
  android:id="@+id/block_3"
  android:layout_width="130sp"
  android:background="#FF884444"
  android:layout_height="wrap_content"
  android:gravity="center"
  android:text="3"/>
```

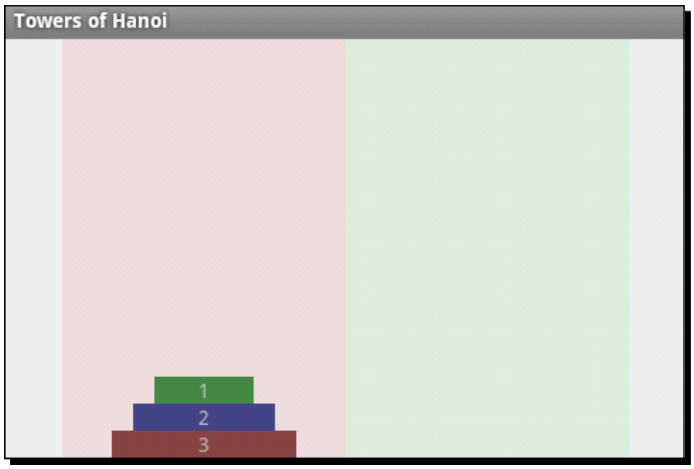
As you can see, these blocks are pretty similar to `block_1`. We've changed the size and the color, but that's about it.

4. Now we can safely remove the old animation that added `block_1` to the scene when we start `HanoiActivity`. Open up `HanoiActivity.java`, and in `onCreate()`, delete the following lines:

```
Animation drop =
  AnimationUtils.loadAnimation(this, R.anim.block_drop);
block1.startAnimation(drop);
```

   Because we are using a `LayoutAnimation`, our start animation is entirely declared in XML, with no Java required! However, we now have new blocks that the user will want to move about too.

5. Let's build and launch the `HanoiActivity` and take a look at it.



What just happened?

Here we used a tween to build up the initial scene that the user will see. The `LayoutAnimation` is simply a wrapper around an animation that you've already defined; it's a simple operation to add one. Layout animations run without requiring any Java code.

Receiving animation events

Our `block_1` animation flies up into the air with no problems, but what about the landing? Right now, it seems to magically reappear where it started, but we want it to come down the tower on the right-hand side. We need some way to remove the block from the left tower and add it to the right tower.

Fortunately for us, Android provides an event-driven way to do things once an animation has finished. Let's take a look at how to add an animation listener to do something.

Time for action – receiving animation events

We want to complete our block's journey from one tower to another. To do this, we will use an `AnimationListener` that adds the block to the second tower as soon as the animation of it leaving the first tower has ended.

1. We will extend the `BlockMover` to do a drop animation as soon as it has finished removing the block from its tower. Add `Animation.AnimationListener` to the interfaces that the `BlockMover` supports.

```
private class BlockMover
  implements Animation.AnimationListener {
```

Also add the new methods that the `AnimationListener` interface provides. Note that we only really care about the last method for this tutorial.

```
  public void onAnimationRepeat(Animation animation) {}
  public void onAnimationStart(Animation animation) {}
  public void onAnimationEnd(Animation animation) {
    // We will fill this in in a minute
 }
```

2. We will associate the `AnimationListener` with the `removeAnimation`, so that we can tell when the block has been removed from the scene. In the bottom of the `move()` method, add this line:

```
removeAnimation.setAnimationListener(this);
```

Now to fill in the contents of `onAnimationEnd()`. It's quite long, so I will interrupt it as we go along, but you should enter all the code in the order it's written.

```
  public void onAnimationEnd(Animation animation) {
    block.post(new Runnable() {
      public void run() {
```

Remember that we should always do graphical stuff in the GUI thread! In this case, it is important that the `removeView()` and `addView()` calls are done there.

```
      ViewGroup toTower = (ViewGroup) findViewById(towers[to]);
      ViewGroup fromTower = (ViewGroup) block.getParent();
      fromTower.removeView(block);
      fromTower.clearDisappearingChildren();
```

We have removed the block from the `fromTower` and it will no longer be drawn on that tower. Note that we also call `fromTower.clearDisappearingChildren()` to remove the animations associated with the block. If we did not do this, you would still see ghostly animations appearing in the empty tower!

```
       toTower.addView(block);
```

```
Animation addAnimation =
  AnimationUtils.loadAnimation(HanoiActivity.this,
  R.anim.block_drop);
block.setAnimation(addAnimation);
```

Here we are adding the block to the second tower, and introducing it to the display by using the `block_drop` animation that we defined earlier.

Finally, we give the block a new `OnClickListener`, to make it bounce back to the place where it came from.

3. There! Now your block should bounce from one tower to the next. Build the code and launch it on your Android device.

What just happened?

By using the `AnimationListener`, we were able to launch a new animation as soon as the first animation had completed.

We also saw that, when moving one animation to another place, we needed to call `clearDisappearingChildren()` on the previous owner, so that it does not receive animation events from the block in its new home. If you do not do this, the user will see ghost animations of your views appearing in the `ViewGroup` from which they have been removed, whenever you call an animation on that view.

Pop quiz – AnimationListeners

1. What should you make sure you do when modifying graphics code in an `AnimationListener` method?
   a. Clear out any old animations with `clearDisappearingChildren()`.
   b. Post the graphics calls to the GUI thread.
2. Which of these is not a method in `AnimationListener`?
   a. `onAnimationStart`
   b. `onAnimationRestart`
   c. `onAnimationRepeat`
   d. `onAnimationEnd`

Interpolating animations

These animations look a bit rigid, don't they? They just move from one place to the next robotically, but most real-world objects have a bit of rhythm, due to the physics underpinning the way they are moved. Let's make our blocks move as if they were real-world objects, by adjusting the timing of their tweens.

Think back to when we were defining animations; do you remember seeing the line `android:interpolator="@android:anim/linear_interpolator"` when we were defining the translate and rotate tweens? This interpolator just moves the object from the start point to the end point (or rotation) at a steady rate.

Next, we are going to use some different interpolators to achieve a more natural animation effect.

Time for action – changing the rhythm with interpolators

We will put a few different interpolators into the mix now, and you can see for yourself which ones look most natural.

1. Open up the file `res/anim/block_drop.xml` and look for the line which says:

```
android:interpolator="@android:anim/linear_interpolator">
```

Change it to read

```
android:interpolator="@android:anim/bounce_interpolator">
```

Now, whenever the block falls into a tower, it will bounce rather than just slide down.

2. Build and run the `HanoiActivity`. What do you think? Does it look more natural now? Can you think of any improvements that we could make to the way it moves?

3. Next, let's do the same to the `block_move_right.xml` animation. Open it up and we'll make some changes. This time, we are going to set different interpolations on different portions of the tween. Let's tell Android of our intentions by replacing the line which says:

   ```
   android:interpolator="@android:anim/linear_interpolator">
   ```

   Replace this line with one that says:

   ```
   android:shareInterpolator="false">
   ```

   Here, we are saying that we do not want there to be one overarching interpolator for all our tweens, and that sub-elements of this `<set>` will be specifying their own interpolators.

4. Now to change the individual interpolators. Firstly, let's make the rotation speed up as it gets further off screen. We do this with an `AccelerateInterpolator`. In the attributes for our `<rotate>` tween, add the following line:

   ```
   android:interpolator="@android:anim/accelerate_interpolator"
   ```

   As you can see, it's the same syntax as before, but it is only associated with one portion of the tween style.

5. When the block is lifted up, let's make it, so that it speeds up at first, but slows down again as it nears the top of the screen. Fortunately for us, there is an interpolator called `AccelerateDecelerateInterpolator` that will do exactly that.

   As you just did for the `<rotate>` tween, add the following line to the first of the two translates.

   ```
   android:interpolator=
     "@android:anim/accelerate_decelerate_interpolator"
   ```

6. Now build and run your activity. You should see a bit more variance in the way that everything moves around.

What just happened?

Interpolators are a feature of animation that are easy to swap around and change, and Android provides us with a palette of useful interpolators to save us from having to write our own.

We took some of the interpolators that we were using and gave them a little bit more character! The look and feel of an animation is sometimes a matter of personal style, so you may prefer it the way it was before.

Using the interpolators provided by Android

Each interpolator adds some characteristic motion to a tween. When we talk about motion in this context, it may literally mean motion (in the rotation and translation tweens) or it may mean the metaphorical motion from the start state to the end state (in the alpha and scale tweens).

Here is a short summary of the interpolators that the Android platform provides. For ease of reference, I've added the reference ID for adding an Interpolator in XML as well as the Java class name in the `android.view.animation` package. This will be handy when we start creating animations in Java in the next chapter.

Linear interpolator

Goes from start to finish at a steady rate. This is the default interpolator - if you don't specify one, then Android assumes that this is what you want.

Java class Name	`LinearInterpolator`
Android reference ID	`@android:anim/linear_interpolator`

Accelerate interpolator

Moves slowly at first, but gathers speed at a linear rate.

Java class Name	`AccelerateInterpolator`
Android reference ID	`@android:anim/accelerate_interpolator`

Decelerate interpolator

The opposite of an accelerate interpolator; it moves quickly at first but loses speed.

Java class Name	`DecelerateInterpolator`
Android reference ID	`@android:anim/decelerate_interpolator`

Accelerate-decelerate interpolator

Performs like an accelerate interpolator until the object reaches midway in its journey. Then it behaves like a decelerate interpolator until it reaches its end point.

Java class name	`AccelerateDecelerateInterpolator`
Android reference ID	`@android:anim/accelerate_decelerate_interpolator`

Bounce interpolator

This interpolator mimics the motion of a ball bouncing.

Java class Name	`BounceInterpolator`
Android reference ID	`@android:anim/bounce_interpolator`

Anticipate interpolator

An anticipate interpolator pulls back before launching – a little bit like a catapult action.

Java class Name	`AnticipateInterpolator`
Android reference ID	`@android:anim/anticipate_interpolator`

Overshoot interpolator

This mimics the action of going too far and then being pulled back into position.

Java class Name	`OvershootInterpolator`
Android reference ID	`@android:anim/overshoot_interpolator`

Anticipate overshoot interpolator

Combines the anticipate and overshoot interpolators to give the animation a springy feel.

Java class Name	`AnticipateOvershootInterpolator`
Android reference ID	`@android:anim/anticipate_overshoot_interpolator`

Cycle interpolator

Moves back and forth around the start point, as if orbiting it.

Java class Name	`CycleInterpolator`
Android reference ID	`@android:anim/cycle_interpolator`

Sharing interpolators

An `AnimationSet` can cause all of its children to use the same interpolator, or it can allow them to define their own instances of interpolators. This is specified in the `android:sharedInterpolator` XML attribute, as you will now see.

android:sharedInterpolator="true"

When using several tweens within a `<set>`, you may wish to define a shared interpolator for all of its child tweens to make use of. This guarantees that they all move in the same style as each other, even if their animations are offset or repeated.

android:sharedInterpolator="false"

Alternatively, you may want some parts of the tween to have a different rhythm to other parts. In this instance, the rotate and lift elements had different physical styles.

`<set>` elements default to having `android:sharedInterpolator="true"`.

If you are making an animation with several interpolators, and you're wondering why your non-shared interpolator is not doing anything, make sure that you explicitly declare `android:sharedInterpolator="false"` in any `<set>` element that you might have in your tween.

Creating and parameterizing interpolators

When you create an interpolator in XML, there is not much you can do to modify its behavior. However, interpolators can also be created in Java, and some of them provide additional parameters in their constructors. These additional parameters can be used to give greater control over how the interpolation looks.

We can take this further, and provide completely new interpolators for use with tween animations. All of the interpolators that we have seen have been defined for us, but it is equally possible to implement the interface `android.view.animation.Interpolator` ourselves and define a new interpolation behavior.

We will learn more about these topics in *Chapter 5*, *Creating Classes for Tween Animation*.

Pop quiz – interpolators

1. Why would you want to use a non-shared interpolator?
    a. To save memory
    b. To apply different motion attributes to different parts of the tween
    c. To apply a consistent motion to the whole tween

2. Which interpolator most closely resembles a catapult pulling back before launching?
   a. `AnticipateInterpolator`
   b. `AccelerateInterpolator`
   c. `BounceInterpolator`
3. Which interpolator orbits back and forth around its start point?
   a. `AccelerateDecelerateInterpolator`
   b. `AnticipateInterpolator`
   c. `CycleInterpolator`

Have a go hero – having fun with interpolators

Now that you've seen the interpolator classes in Android, perhaps you think that maybe one or two of them would be better than the ones I showed you in the example. Try changing the interpolators in the `HanoiActivity` to make them more exciting. Get a feel for how they all look, and try combining them with each other.

Personally, I think that the `AnticipateInterpolator` looks good in the `block_move_*` animations, catapulting the block into the sky.

Finding out more

You can get more information on tween animations, and also the animation class, from the Android developer guide's *Animation Resources* section.

`http://developer.android.com/guide/topics/resources/animation-resource.html`

This covers some of the information in this chapter, and also an introduction to some other techniques that we will have a look at in the next chapter. You may also find the package index for `android.view.animation` useful:

`http://developer.android.com/reference/android/view/animation/package-summary.html`

This contains a list of all of the standard animations and interpolators that Android provides for us.

Summary

Congratulations! You have finished making an animated Towers of Hanoi game! On the way, you have learned all of these things:

- The building blocks of a tween: alpha, translate, rotate, and scale
- How to stop and start an animation
- Sequencing several building blocks to make a complex animation
- Receiving events from an animation
- Interpolators, and why they're useful

By the way, here are the rules to the game: move all the blocks, so that the tower is on the right-hand side of the screen instead of the left. Have fun!

Now that we've had an introduction to tween animations, the next chapter will be about animators. We will also cover some fancier techniques that apply to tween animations.

4
Animating Properties and Tweening Pages

In the last chapter, we saw some basic view-based animation techniques and how they could be parameterized and combined to create a natural animation style.

In this chapter, we will build on the tweening techniques we've already learned, and also apply some new techniques that were introduced in Android 3.0.

In this chapter, we shall:

- Use a `ViewFlipper` for animating a book-like application
- Use Java to define a new tween animation and apply it to a view
- Use an `ObjectAnimator` to apply an animation to a view, a bit like a tween
- Use a `ValueAnimator` to generate values, which we will use for a more complex animation
- Compare the `Animator` classes to the tween classes from the previous chapter

So let's get on with it...

Note for developers using versions of Android before 3.0

So far, everything we have learned has been backwards-compatible with previous versions of Android. This will hold true for the first part of this chapter, but not the second. That is to say that `ViewFlippers` are backwards-compatible with previous versions of Android, but `ValueAnimators` and `ObjectAnimators` are new to version 3.0.

At the time of writing (mid-2011), the Android Compatibility Package does not help with this problem.

Turning pages with a ViewFlipper

`ViewFlipper` is a neat little wrapper class for applying a **page-turning** animation to a set of pages. It makes use of the tween animation classes that we learned about in the previous chapter, and extends them with an XML interface.

The `ViewFlipper` is actually a subclass of something called a `ViewAnimator`. Do not get confused! A `ViewAnimator` is a completely different class to a `ValueAnimator` or an `ObjectAnimator`, and they are not interchangeable.

Let's see more.

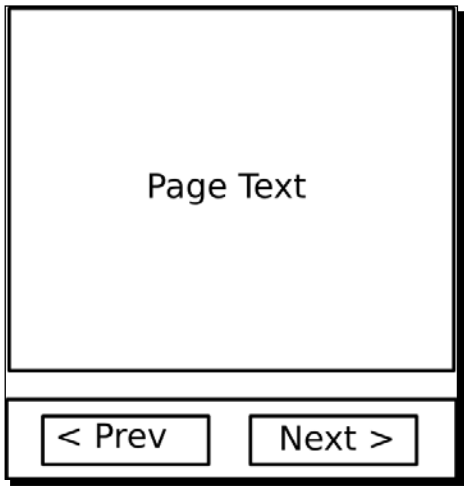
Time for action – making an interactive book

You have been hired by a children's book publisher to make an interactive book. The book will teach kindergarten children about different sorts of motion by showing them small animations on the pages.

First up, we will use a `ViewFlipper` widget to make an animated page-turning interface. What better way to learn about a page-turning widget than by using it to make a book? We will also add some simple pages to test the `ViewFlipper`, which we can add animations to in some later examples.

1. Create a new Android project with the following settings:
    - **Project name**: `Interactive Book`
    - **Build target**: `Android 3.0`
    - **Application name**: `Interactive Book`
    - **Package name**: `com.packt.animation.interactivebook`
    - **Activity**: `InteractiveBook`

The first thing we will do is to define a layout for our book. We want it to look a little bit like the following screenshot:



2. So let's begin! Open `res/layout/main.xml` and create the following layout:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent">
    <ViewFlipper android:id="@+id/pages"
      android:layout_width="fill_parent"
      android:layout_height="fill_parent"
      android:layout_weight="2">
    </ViewFlipper>
    <LinearLayout
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      android:layout_weight="1"
      android:gravity="center"
    >
      <Button
        android:id="@+id/prev"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_weight="1"
        android:drawableLeft="@android:drawable/ic_media_previous"
        android:text="Previous" />
```

```
      <Button
        android:id="@+id/next"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_weight="1"
        android:drawableRight="@android:drawable/ic_media_next"
        android:text="Next" />
    </LinearLayout>
  </LinearLayout>
```

Here we have set up the layout of the application, but we have not yet added any pages. In XML, the pages of the `ViewFlipper` are created by adding child layouts to `ViewFlipper`.

3. Firstly, we will want a Drawable, which we can animate. Create a new file in `res/drawable` called `res/drawable/ball.xml` and give it the following contents:

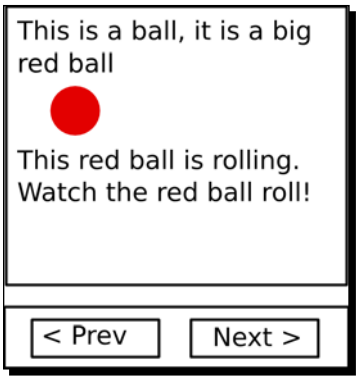
```
<?xml version="1.0" encoding="utf-8"?>
<shape
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:shape="oval" >
  <gradient
    android:startColor="#FFFF0000"
    android:endColor="#FF551010"
    android:angle="270"/>
  <size
    android:height="40dp"
    android:width="40dp"/>
</shape>
```

This is just an ordinary `ShapeDrawable`; there's no special animation or anything here! We will just use it as a simple ball graphic while we are writing the book. Later on, we will add animation.

4. In `main.xml`, between the `<ViewFlipper>` and `</ViewFlipper>` tags, add the following new elements:

I will intersperse the code with pictures, so that you can see what we are adding as we go along. You should add the XML in order, and use the pictures as a quick guide to get what you want?

First, take a look at the following screenshot. This should give you an idea of the structure of the page that we are going to make:

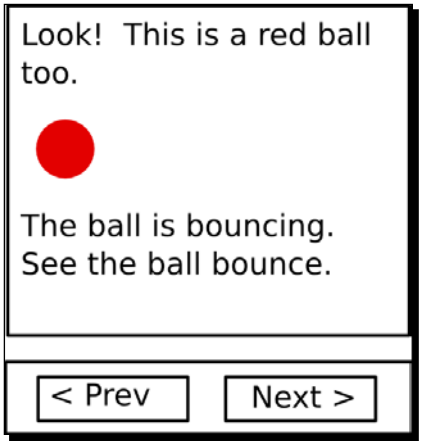


Looks simple enough. Let's write the layout code for it.

Remember that this is going between the `<ViewFlipper>` and `</ViewFlipper>` tags.

```
<LinearLayout
  android:layout_width="fill_parent"
  android:layout_height="wrap_content"
  android:orientation="vertical">
  <TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="This is a ball, it is a big red ball"
  />
  <ImageView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/rollingball"
    android:src="@drawable/ball"
    android:paddingLeft="60dp"
  />
  <TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text=
      "This red ball is rolling.  Watch the red ball roll!"
  />
</LinearLayout>
```

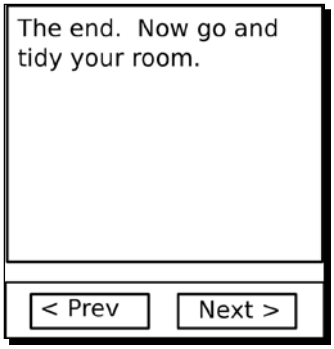
That was *page 1*, now let us make *page 2*. It will be laid out as in the next screenshot:



The layout text that follows should go between the `<LinearLayout>` for *page 1* and the `</ViewFlipper>` tag.

```
<LinearLayout
  android:layout_width="fill_parent"
  android:layout_height="wrap_content"
  android:orientation="vertical">
  <TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Look!  This is a red ball too."
  />
  <ImageView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/bouncingball"
    android:src="@drawable/ball"
    android:paddingLeft="60dp"
  />
  <TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text=
    "The ball is bouncing.  See the ball bounce."
  />
</LinearLayout>
```

Finally, this is what the last page will look like:



As you might suppose, the layout that follows goes between *page 2* and the `</ViewFlipper>` tag.

```
<TextView
  android:layout_width="fill_parent"
  android:layout_height="wrap_content"
  android:text="The end.  Now go and tidy your room."
/>
```

Our content pages are defined in XML. Our `ViewFlipper` is going to treat each of the highest-level elements (the `LinearLayout` and the `TextView`) as pages in its layout. In this sense, it works exactly as a `FrameLayout` would work.

5. Okay, great. If you ran this now, you would be able to see the first page, but we've still not connected the page-turning buttons. Let's do that now. Open up `InteractiveBook.java` and add the following import declarations:

```
import android.view.View;
import android.widget.Button;
import android.widget.ViewAnimator;
```

The last one is the most important. As I mentioned earlier, the `ViewFlipper` is a subclass of `ViewAnimator`. Seeing, as we don't need to use any of the methods of the subclass, we are only going to work with its superclass.

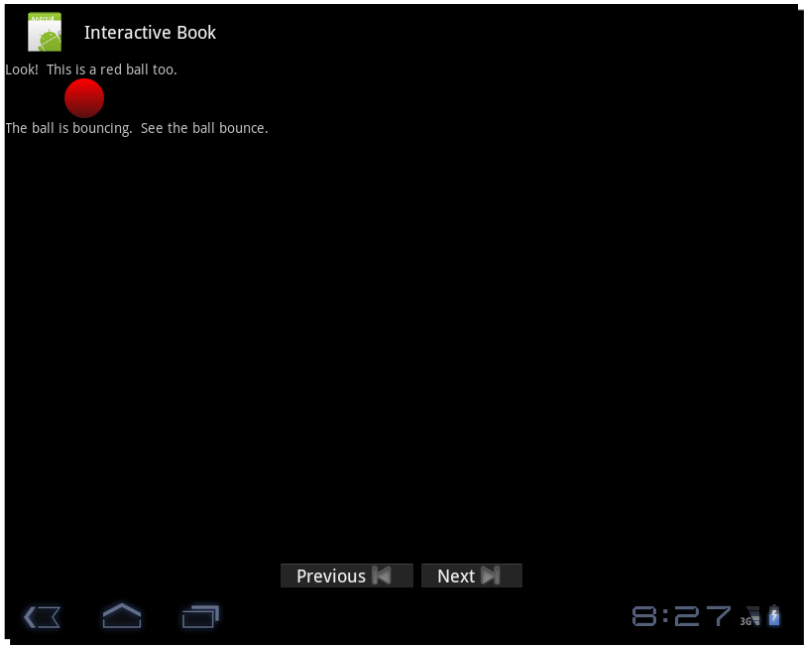
6. Now, add the following block of code at the end of `onCreate()`.

```
final ViewAnimator pages =
  (ViewAnimator) findViewById(R.id.pages);
Button prev = (Button) findViewById (R.id.prev);
Button next = (Button) findViewById (R.id.next);
prev.setOnClickListener(new View.OnClickListener() {
  public void onClick (View v) {
    pages.showPrevious();
```

```
    }
});
next.setOnClickListener(new View.OnClickListener() {
  public void onClick(View v) {
    pages.showNext();
  }
});
```

Here we can see exactly how to write a page-turning control in a `ViewFlipper`. Simply call `pages.showPrevious()` or `pages.showNext()`.

7. Build and run your application. You should see that the `ViewFlipper` turns pages perfectly well now.



There's something missing from this interactive book—the animation between the pages is not very smooth. In fact, all it does is switch between one page and the next.

Let's give it a more natural feel with a page turning animation.

In `res/anim`, create a new XML file called `slidein.xml`. This will be an ordinary tween animation, which is similar to the one in the previous chapter. We will use this animation to introduce new pages to the screen.

Add the following block of code to it:

```
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android="http://schemas.android.com/apk/res/android"
  android:interpolator="@android:anim/decelerate_interpolator">
  <translate
    android:fromXDelta="100%p"
    android:toXDelta="0"
    android:duration="500"
  />
</set>
```

This means that when the user turns a page, the new page comes across from the right-hand side of the screen, as if they were turning pages in a book (sort of).

8. Now let's add the opposite effect, by removing the old page from the screen. In `res/anim`, create another XML file called – you guessed it – `slideout.xml`.

In it, add the following XML:

```
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android="http://schemas.android.com/apk/res/android"
  android:interpolator="@android:anim/accelerate_interpolator">
  <translate
    android:toXDelta="-100%p"
    android:fromXDelta="0"
    android:duration="500"
  />
</set>
```

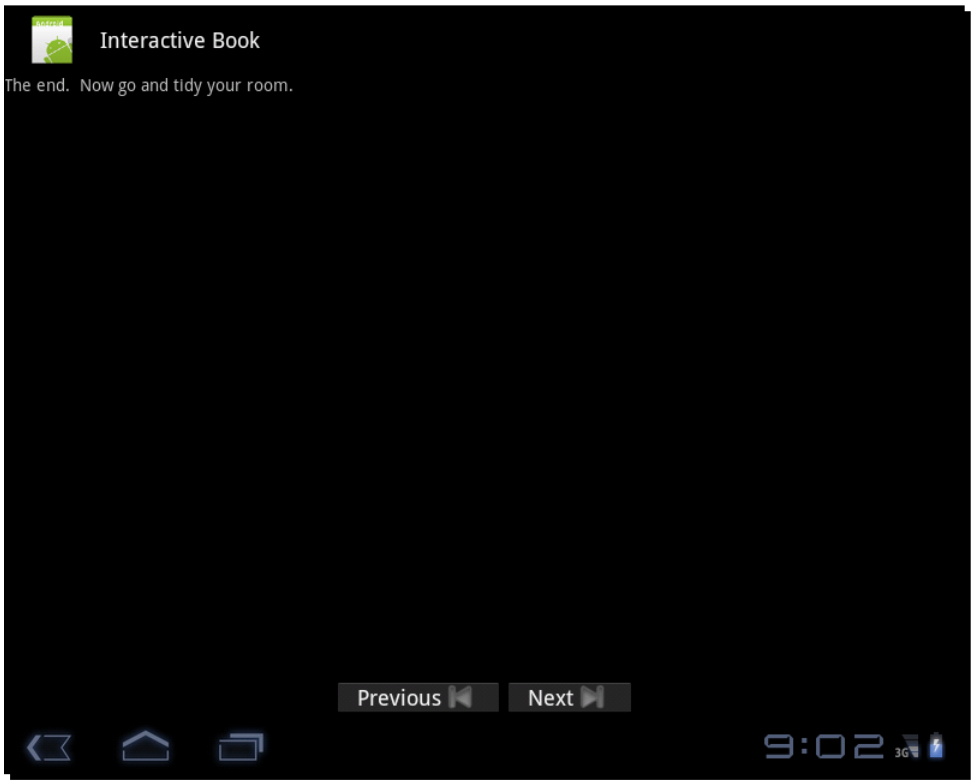
As the pages arrive from the right, they also move off to the left.

9. Now we need to add this animation to the `ViewFlipper`. Open up `main.xml` again, and add these attributes to our declaration of the `ViewFlipper`.

```
<ViewFlipper android:id="@+id/pages"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  android:layout_weight="2"
  android:inAnimation="@anim/slidein"
  android:outAnimation="@anim/slideout" >
```

This is very similar to how we added tween animations to elements in the previous chapter, but this time you can see that there are two different animations.

10. Now build and run the interactive book. You will see that your pages now transition smoothly from one to the next.



What just happened?

We created a book-like application that displays several pages of information. We created a new `ViewFlipper` widget and applied a page-turning animation to it to give it a natural, book-like feel.

For convenience, the animations applied to `ViewFlipper` will apply to every single page that is contained within it. Remember, you do not need to apply an individual tween to each page in your book. Just adding the `inAnimation` and `outAnimation` in your `ViewFlipper` will be sufficient.

Pop quiz – ViewFlippers

1. How do `ViewFlippers` let you define pages in XML?
   a. As individual XML files.
   b. As child elements of the `ViewFlipper`.
   c. You have to add them through Java.
2. How do you make a page-turning animation for a `ViewFlipper`?
   a. Make a tween animation.
   b. Define a `TransitionDrawable`.
   c. Write a frame animation.
3. How do you apply an animation to a `ViewFlipper`?
   a. Write a Java loop that applies the animation to each page.
   b. Reference it in the `ViewFlipper's android:inAnimation` and `android:outAnimation` attributes.
   c. Reference it in the `ViewFlipper's android:animation` attribute.

Have a go hero – improving the ViewFlipper

Think about how you would like to turn pages in a book. Perhaps the motion that we created above could be improved in some way.

Edit the `slidein.xml` and `slideout.xml` tween animations, and create a new animation of your own invention.

Take a look at Chapter 3, *Tweening and Using Animators*, to get some inspiration for animations that you could try.

Creating tween animations in Java

So far, all of the tween animations that we made have been created in XML, and there is good reason for this. Why should you want to clutter up your logical code with a load of presentation code?

But sometimes you want to create your tweens programmatically, perhaps because they rely on some computed values or it makes sense to describe them computationally.

Whatever the reason, we can use Java to create tween animations just as easily as we can create them in XML.

Time for action – creating a tween in Java

We want to make a new animation to replace `slidein.xml`. This time, we want our pages to come in from the right, as before, but we will add a scale animation too, to make it look more exciting. It will be as if the page is being pulled from a tall stack of pages, just out of view.

But we're bored of XML. Don't ask me why, perhaps it's because of all those pointy brackets. Give us the round parentheses of Java, we say! We will use the Java equivalent of the XML tags for `<set>`, `<translate>`, and `<scale>` animations.

1. Open up `InteractiveBook.java` and add the following import lines:

```
import android.view.animation.Animation;
import android.view.animation.AnimationSet;
import android.view.animation.ScaleAnimation;
import android.view.animation.TranslateAnimation;
```

   All these classes describe animations like the ones we made use of in XML, in the previous chapter.

   - `<set>` becomes `AnimationSet`
   - `<scale>` becomes `ScaleAnimation`
   - `<translate>` becomes `TranslateAnimation`

2. Next, let's construct an `AnimationSet`, in which we can build a compound animation. Navigate to the bottom of the `onCreate()` method and add the following code:

```
AnimationSet slideAndScale = new AnimationSet(true);
```

   This creates an `AnimationSet`. The Boolean `true` means that we want a shared interpolator. Recall the previous chapter; it's the Java equivalent of writing the following in XML (don't add this to your code!):

```
<set xmlns:android="http://schemas.android.com/apk/res/android"
  android:shareInterpolator="true">
```

3. Now to create a translate animation, go into the `<set>`. Add the following code below our `AnimationSet`.

```
TranslateAnimation slide = new TranslateAnimation(
  Animation.RELATIVE_TO_PARENT, 1f,
  Animation.RELATIVE_TO_PARENT, 0,
  Animation.RELATIVE_TO_SELF, 0,
  Animation.RELATIVE_TO_SELF, 0
);
```

The Java constructor for a `TranslateAnimation` lets you specify the `fromX`, `toX`, `fromY`, and `toY` components of the translation. The enumeration values are the equivalent of the different value types that you can input in XML.

The options that you can specify are `RELATIVE_TO_PARENT`, `RELATIVE_TO_SELF`, and `ABSOLUTE`.

4. Now to make a scaling animation.

```
ScaleAnimation scale = new ScaleAnimation(
  10,
  1,
  10,
  1
);
```

Similar to the `TranslateAnimation` constructor, the arguments are `fromX`, `toX`, `fromY`, and `toY`, except that this time they are all floating-point multiplier values.

5. Now we add them in to the main `AnimationSet` as follows:

```
slideAndScale.addAnimation(slide);
slideAndScale.addAnimation(scale);
```

6. Next, we want to specify the duration of the animation. As everything has been already added to the `AnimationSet`, all we need to do is add the following line:

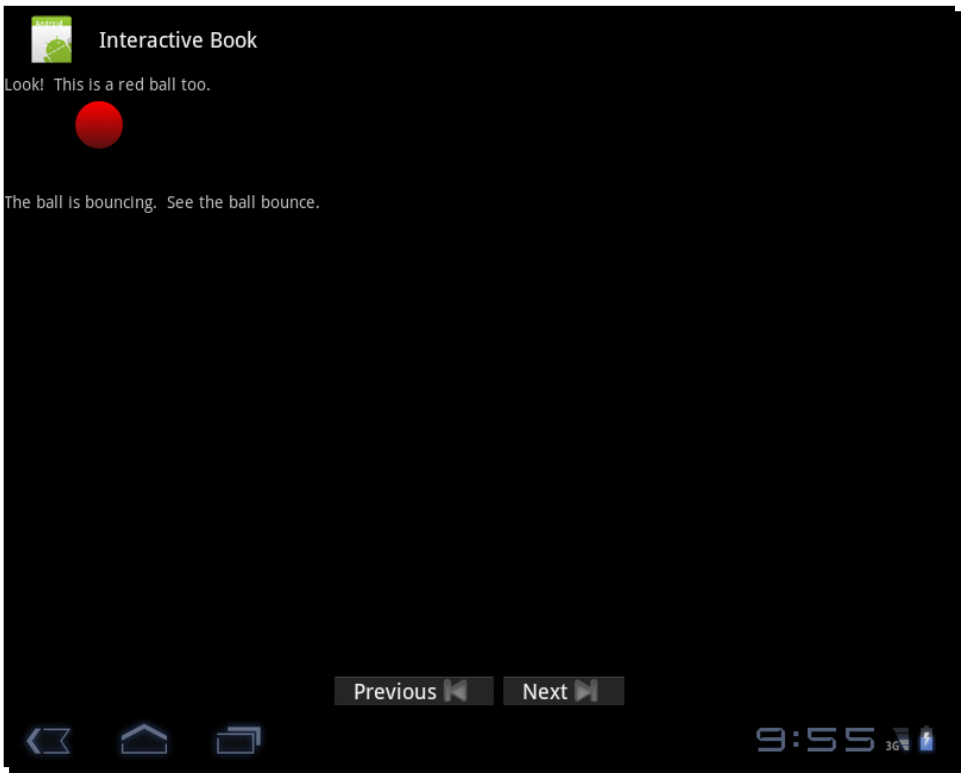
```
slideAndScale.setDuration(1000);
```

As you probably expect by now, `1000` is the duration in milliseconds to show the animation.

7. This concludes the construction of the `AnimationSet`. So all we need to do now is to set it as the `inAnimation` on our `ViewFlipper`. We've already got access to the `ViewFlipper` object as pages, so we can simply add this:

```
pages.setInAnimation(slideAndScale);
```

8. There! Build and run your activity and observe the new animation.



As you can see, the image now scales as the page is turned.

What just happened?

We've created a new page-turning animation, which is an `AnimationSet` containing a `ScaleAnimation` and a `TranslateAnimation`. Now the page looks like it is being lifted into view, as it is turned.

We've created tween animations before, but this one was in Java. We have seen that it is possible to create a tween animation in Java that provides the same sort of functionality, which you would expect from a tween animation created in XML. By comparing the source against its equivalent in XML, you can see where the differences lie.

Writing the SlideAndScale animation in Java

In Java, we instantiate the `AnimationSet`, `ScaleAnimation`, and `TranslateAnimation`. The animation objects are parameterized in their respective constructors.

We then add the `ScaleAnimation` and `TranslateAnimation` to the `AnimationSet`.

```
AnimationSet slideAndScale = new AnimationSet(true);
TranslateAnimation slide = new TranslateAnimation(
  Animation.RELATIVE_TO_PARENT, 1f,
  Animation.RELATIVE_TO_PARENT, 0,
  Animation.RELATIVE_TO_SELF, 0,
  Animation.RELATIVE_TO_SELF, 0
);
ScaleAnimation scale = new ScaleAnimation(
  10,
  1,
  10,
  1
);
slideAndScale.addAnimation(slide);
slideAndScale.addAnimation(scale);
slideAndScale.setDuration(1000);
```

Writing the SlideAndScale animation In XML

In XML, the tween animation is created by declaring a `<set>` tag that contains the `translate` and `scale` operations as child nodes. The animations are parameterized by giving them attributes.

```
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android="http://schemas.android.com/apk/res/android"
  android:shareInterpolator="true"
  android:duration="500">
  <translate
    android:fromXDelta="100%p"
    android:toXDelta="0"
  />
  <scale
    android:fromXScale="10"
    android:toXScale="1"
    android:fromYScale="10"
    android:toYScale="1"
   />
</set>
```

As you can see, the advantage of the XML version is that it is more clearly laid-out. This is not just a matter of personal taste; by writing each attribute name as you assign it, there is never any ambiguity as to which value you are assigning. Look at the Java version and see if you can remember what order the constructor arguments are constructed in. It's hard, isn't it?

In conclusion, programmatic tween creation should only be used when you can think of a clear advantage.

Pop quiz – Java tweens

1. How would you set the duration of a tween animation in Java?
   a. `Animation.setSpeed(int)`.
   b. `Animation.setDuration(int)`.
   c. `Animation.addAnimation(Animation)`.
2. You've seen two primitive animation types that have been used in Java, what do you think the `<rotate>` animation class is called in Java?
   a. `TurnAnimation`.
   b. `SpinRoundAnimation`.
   c. `RotateAnimation`.

Have a go hero – tweening using Java

Okay, now that we've made a tween that scales and slides in the graphic, have a go at making a similar tween for the `outAnimation` part of the interactive book.

Look at the code you've already written, and make a new animation in Java with the following properties:

- As the page leaves the screen, it moves to the left
- As the page leaves the screen, it gets larger

Animating with ObjectAnimator

`ObjectAnimators` are the first animations you will learn about which are new to Android 3.0. Recall that tweens are all about moving views from one place to another, and that they describe different kinds of motion. Unlike tweens, animators work by updating values on an object in a much more programmatic way.

Animators just change numeric parameters on an object that they know nothing about, for instance, translating a view from one place to another. But by applying an animator to the `X` and `Y` coordinates of a view, an `Animator` can be used to perform the same task that a tween would do.

Which one you choose is up to you. Animators give you more flexibility, but they might not be as clear to read.

Time for action – animating the rolling ball

The children's book company has got back to us and they're not happy with the first page of our interactive book. It says that the ball is rolling, but it isn't!

We're going to fix this by using an `ObjectAnimator` to move the ball backwards and forwards across the screen.

1. Open up `InteractiveBook.java` and add a new import declaration:

   ```
   import android.animation.ObjectAnimator;
   ```

   This should come as no surprise! I already told you that we would be using the `ObjectAnimator` class.

2. Next, go to the end of the `onCreate()` method, and add the following lines:

   ```
   View rollingBall = findViewById(R.id.rollingball);
   ObjectAnimator ballRoller =
     ObjectAnimator.ofFloat(
       rollingBall,
       "TranslationX",
       0,
       400
     );
   ```

3. Underneath the code you just added, add the following.

   ```
   ballRoller.setDuration(2000);
   ballRoller.setRepeatMode(ObjectAnimator.REVERSE);
   ballRoller.setRepeatCount(ObjectAnimator.INFINITE);
   ```

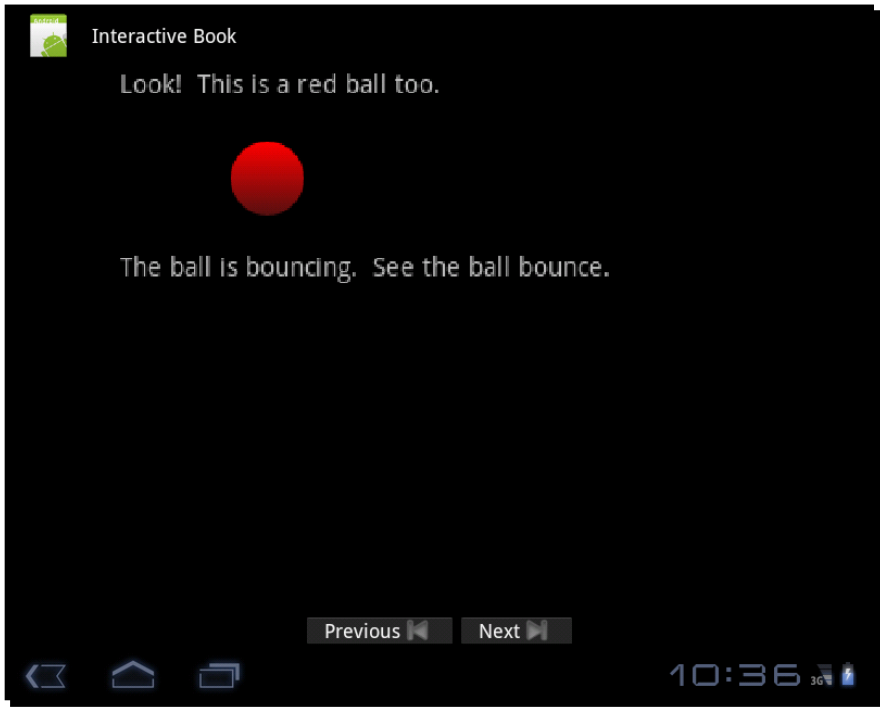
   These terms should look familiar to the XML we wrote in the previous chapter, although the Java form will seem unfamiliar.

   - `setDuration` sets the duration of the animation in milliseconds
   - `setRepeatMode` can be either `REPEAT` or `REVERSE`
   - `setRepeatCount` can be an integer number of repeats or (as it is here) `INFINITE`

4. One last line you need to add after all this is to tell Android to begin the animation immediately.

```
ballRoller.start();
```

5. And that's it! Couldn't be simpler. Build and run your activity and you will see that the ball on the first page now rolls backwards and forwards.



This red ball is rolling, because we animated it!

What just happened?

Here we used our first `Animator`, and it is an `ObjectAnimator`. The `ObjectAnimator` provides a simple way to animate a scene by continuously updating a parameter of that scene.

The `ObjectAnimator` takes several arguments of different types. Let's take a look at them in a bit more detail.

Constructing ObjectAnimators

You should always use a factory method to construct your `ObjectAnimators`. This means any one of the following:

- `ObjectAnimator.ofInt`
- `ObjectAnimator.ofFloat`
- `ObjectAnimator.ofObject`
- `ObjectAnimator.ofPropertyValuesHolder`

The second two options are more complex than the first two. For now, we will only concern ourselves with parameters that can be manipulated using `int` or `float` types.

The first parameter of the `ObjectAnimator` factory method is the object on which it will be operating. This neatly associates the animator with that object immediately, and you don't need to call a separate method to associate the two, as we did with `setAnimation` in the previous chapter.

The second parameter is the name of the parameter that you want to modify. See the next little section for a bit more information about that.

The remaining parameters are a list of values between which the animator will interpolate. We used two for this example, but it could be any length list. Let's examine the way in which we constructed the `ObjectAnimator` in the previous example.

Breaking down the construction of ballRoller

If you look at *step 2* of the example, you will notice that the `ObjectAnimator` itself is created by the factory method. There are other factory methods for different types of data, but we are interested in floating point values.

We want to animate the `TranslationX` parameter, which is an ordinary parameter that all view classes have. An `ObjectAnimator` can animate any parameter that can be accessed with a setter method. The `TranslationX` parameter takes a floating point value; hence the use of the `ofFloat` factory method earlier.

The values `0` and `400` are the range of animation. The parameter `TranslationX` will be animated, by giving it a sequence of floating point inputs between these two values.

Finally, notice that the factory method takes the object that it will be animating as an argument.

Getting and setting with ObjectAnimators

The `ObjectAnimator` works by calling `setParameterName(value)` repeatedly on an object. The parameter that you specify in the factory method will be evaluated in an attempt to discover a setter method on the object, which you passed in.

For instance, let's say that you had an object `fishTank` of class `FishTank`, and an integer method `setOctopuses(int numberOfOctopuses)`. If you wanted to animate the change in the number of octopuses, you could write something like:

```
ObjectAnimator.ofInt
  (fishTank, "Octopuses", minOctopuses, maxOctopuses);
```

Pop quiz – ObjectAnimators

1. How is an `Animator` similar to a tween?
   a. Both are specified in XML.
   b. Both apply to views.
   c. Both describe their animation in terms of motion about the screen.
2. Which of these is passed in when constructing an `ObjectAnimator`?
   a. The object which you intend to animate.
   b. An interpolator.
   c. The activity's context.

Have a go hero – what else can you do with the ball?

Our children's book publisher wants some more pages in their book, and they have asked us if we can think of any more motions or effects that we could give to the ball to make new pages.

Here's what we're going to do:

1. Open up the Android API documentation and go to the page for `android.view.View`.
2. Look for any public setters, which could be animated with an `ObjectAnimator`.
3. In the `Interactive Book`, create a new page with a ball in it, and add this new effect.

If you're lost for direction, think about how you would animate a ball being squeezed.

Animating values with ValueAnimator

`ValueAnimator` is the superclass of `ObjectAnimator`, so it has a lot in common with it. It does not provide the same factory methods that allow you to directly animate an object, but it does have very similar factory methods. It also does not automatically connect to an object; you have to do that yourself.

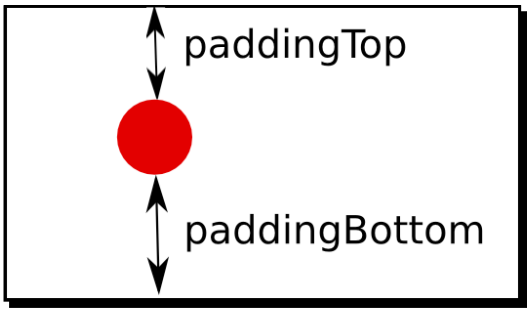
So instead of changing an object, you have to implement a handler that will receive a new value every time the `Animator` performs an update. This could be any sort of class; it doesn't even need to be connected to a view.

Let's learn some more by working with an example.

Time for action – making a ball bounce

Our children's book publisher has come back to us. They thank us kindly for making the ball on the first page roll along correctly, but they are a bit disappointed in *page 2*. *Page 2*, if you recall, is a ball that is supposed to bounce.

We'd better fix that now. Let's use a `ValueAnimator` to change the padding of the ball in the `ImageView`. Look at the following screenshot to see what we're planning:



How the ball-bouncing animation is going to be calculated?

Makes sense? Imagine if, in every frame, we add a little bit to `paddingTop` and take a little bit away from `paddingBottom`. The ball will fall down!

We can also reverse this process to make the ball bounce upward.

1. Open up `InteractiveBook.java`, and add the following new import:

   ```
   import android.animation.ValueAnimator;
   ```

   This is the class we will be working with to animate the bouncing ball.

2. Next, scroll down to the bottom of the `onCreate()` method. We'll be adding a little more there. Firstly, we need to get the ball that we want to bounce.

```
final View bouncingBall = findViewById(R.id.bouncingball);
```

And that's precisely what we've done here.

3. Now let's create our `ValueAnimator`.

```
ValueAnimator ballBouncer = ValueAnimator.ofInt(0,40);
```

Here, we have created the animator that will do the bouncing, but we haven't connected it to anything just yet. This will be an integer animator, with a range between `0` and `40`.

4. Let's get stuck in and add a listener that can receive animation updates.

```
ballBouncer.addUpdateListener(
  new ValueAnimator.AnimatorUpdateListener() {
    public void onAnimationUpdate(ValueAnimator ballBouncer)
      {
      //We'll fill this out in a minute
    }
  }

);
```

This style should look a lot like any other listener you will have created. Note that this one provides us with a `ValueAnimator` as its data.

This means that we have complete access to the `ValueAnimator` behavior from within this `onAnimationUpdate` method.

5. Before we fill out the listener method of the `ballBouncer`, let's set some parameters on the `ballBouncer` itself.

```
ballBouncer.setDuration(2000);
ballBouncer.setRepeatMode(ValueAnimator.REVERSE);
ballBouncer.setRepeatCount(ValueAnimator.INFINITE);
```

Do these look familiar? They're exactly the same as we set on our ball rolling animation in the previous tutorial! As before, we want this ball to reverse its action after it completes. We also want the animation to loop indefinitely.

The duration will be `2000` milliseconds per animation loop.

6. We'd also like to change the frequency with which the `ValueAnimator` updates. `ValueAnimators` have one frame update thread, which acts globally for all `ValueAnimators`, so we must set the update frequency globally for all updates.

```
ValueAnimator.setFrameDelay(50);
```

This specifies the delay in milliseconds between frames; recall that we did something similar when making frame animations.

Now to get stuck into the code for our `AnimatorUpdateListener`. Go back to the line that says:

```
// We'll fill this out in a minute
```

Replace this line with the highlighted code, as follows:

```
public void onAnimationUpdate(ValueAnimator
  ballBouncer) {
final int animatedValue =
(Integer)ballBouncer.getAnimatedValue();
bouncingBall.post(new Runnable() {
  public void run() {
    bouncingBall.setPadding(
      bouncingBall.getPaddingLeft(),
      40 - animatedValue,
      bouncingBall.getPaddingRight(),
      animatedValue);
    bouncingBall.invalidate();
  }
}
);
}
```

Okay, we've done a couple of things here. We grab the `animatedValue` from the `ValueAnimator` that was passed in. Remember how we used the `ObjectAnimator` to set `X` axis values for the rolling ball? This is the same data that was used there, except that a listener rather than a setter method gives it to us.

Next, we apply the values that we have passed in to the padding values in the `bouncingBall` image. Recall the picture at the beginning of this tutorial, and see that we are keeping the left and right padding constant.

Notice also that every time we increase or decrease the top padding, we make a corresponding decrease or increase to the bottom padding.

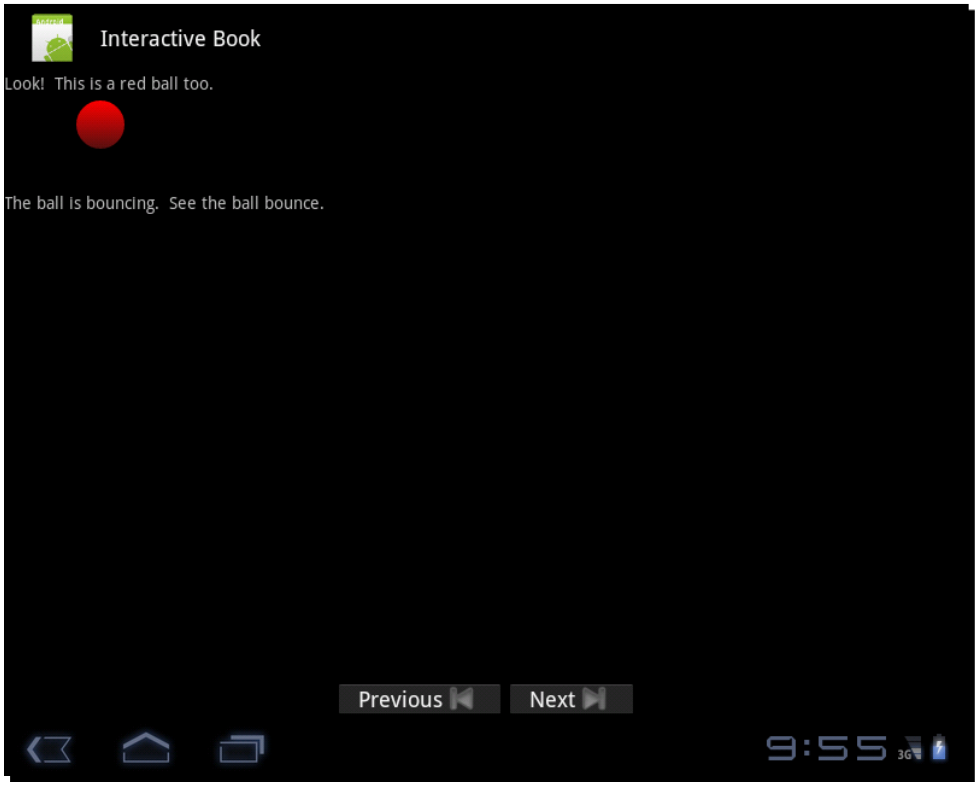
Because this change happens to a GUI element, we are using a `post()` method to ensure that it occurs in the GUI thread.

We also kick off an `invalidate()` at this point to ensure that the GUI will be updated at every animation frame.

7. Okay, we're nearly there! Finally, we need to ensure that the animation is started when the activity starts up. Right at the end of `onCreate()`, add the following line:

```
ballBouncer.start();
```

8. Now build and run the interactive book and see the ball bounce!



You should see something similar to the previous image, where the red ball is moving up and down in a robotic sort of fashion.

What just happened?

In this example, we have made an animation of a bouncing ball, where the bounce animation was provided by a `ValueAnimator`.

As you have seen, the `ValueAnimator` gives a much more flexible interface onto an event-generating Animator.

We created a listener object that received messages from a `ValueAnimator`, did some processing on the information that it received, and then performed an animation activity.

Note that this was quite a contrived way to make a ball bounce. In practice you might want to consider a different approach! The example was chosen to make it easy to observe the application of the `ValueAnimator`.

Updating the frame rate

All `ValueAnimators` (including the `ObjectAnimator` from the previous tutorial) have a common thread, which updates them all. If you want to increase the frame rate of one of the animations, you must also increase it for all the other `ValueAnimator` animations on your display.

The call to update the frame rate is:

```
ValueAnimator.setFrameDelay(50);
```

Changing the interpolator

Like tween animations, `Animators` have an interpolator that specifies the rhythm with which the animation takes place. We can use the same interpolators on animators that we use with tweens.

Let's take a look at this with a simple example.

Time for action – improving our bouncing ball

The bouncing ball looks a little unnatural bouncing like that. Let's specify another interpolator that will make the bouncing action look a little bit more natural.

1. Open up `InteractiveBook.java`. Add another import at the top of the file.

   ```
   import android.view.animation.BounceInterpolator;
   ```

   You might remember that we used one of these in the previous chapter. We want a bouncing action, so we should use a bouncing action, right? I'm not sure, let's try it and see if it works.

2. Next, go to the `onCreate()` method, and find the part where you were specifying parameters for the `ballBouncer`. Add the highlighted line in the following block of code:

   ```
       ballBouncer.setDuration(2000);
       ballBouncer.setRepeatMode(ValueAnimator.REVERSE);
       ballBouncer.setRepeatCount(ValueAnimator.INFINITE);
   ballBouncer.setInterpolator(
   new BounceInterpolator());
   ```

   This is how you apply an interpolator to an `Animator`.

3. Build and run the `Interactive Book` activity; how does it look?

4. Hmm, it's not quite as natural as I'd hoped. It is going the wrong way, and it seems to be bouncing on the ceiling. Let's try a different interpolator.

   In the *import* section of `InteractiveBook.java`, add one more class:

   ```
   import android.view.animation.DecelerateInterpolator;
   ```

5. Next, change the interpolator line to call the new interpolator:

   ```
   ballBouncer.setInterpolator(
     new DecelerateInterpolator());
   ```

6. Now build and run it; much better now!

What just happened?

We have updated the bouncing ball animation to make the bouncing activity look a lot more realistic.

Just as with the interpolators in the tween animations, the look and feel of an `ObjectAnimator` or `ValueAnimator` can be changed dramatically, by changing its interpolator. The interpolators we used in this example to give the bouncing ball a more dynamic feel are the exact same interpolators, which we were using in the previous chapter to move blocks around in the *Towers of Hanoi* game.

You may want to refer to the list of interpolators in the previous chapter to remind yourself of the interpolators that are available.

Pop quiz – ValueAnimators

1. How is a `ValueAnimator` related to an `ObjectAnimator`?
   a. `ValueAnimator` is the superclass of `ObjectAnimator`
   b. `ObjectAnimator` is the superclass of `ValueAnimator`
   c. They are only indirectly related

2. How do you change the rhythm of a `ValueAnimator` animation?
   a. Call `ValueAnimator.setFrameRate()`
   b. Use an interpolator
   c. Use a tween

Comparing animators and tweens

In this chapter, we have created two animators that perform simple motion animations on a view. You could consider doing activities like this using a tween animation, and of course it would work just as well.

The relative benefits of the two are summarized as follows:

Advantages of animators

- They are easy to map to programmatic animations, by adding a setter method or by implementing a listener
- They do not restrict the activities that they can perform on screen to scales, translates, alpha, and rotates
- You can add more than two **keyframes** to an `Animator`

Advantages of tweens

- Tweens are more intuitive; they literally describe what the user will see on the screen
- Tweens can be described in XML

Things that are common between animators and tweens

- Both techniques use the same interpolators
- Both have a start and an end point

Summary

In this chapter, we have made an interactive book using the Android `ViewFlipper` class to make a page-turning interface.

We also learned about the `ValueAnimator` and `ObjectAnimator` classes, which are new animation techniques in Android 3.0.

Specifically, we covered:

- Describing a `ViewFlipper` in XML
- Manipulating the `ViewFlipper` using events in Java
- Specifying a tween animation in Java and comparing it to XML

- Creating `ObjectAnimators` to animate a parameter on an object
- Creating `ValueAnimators` to provide animation events for a listener to use

We considered the relative flexibility of the `ValueAnimator` versus the simplicity of using the `ObjectAnimator` to achieve an animation.

We have started to gain a more in-depth knowledge about animating views. In the next chapter, we will look at more advanced ways of using Java to customize an animation.

5
Creating Classes for Tween Animation

In the last chapter, we developed the idea of having specialized classes for using animations, by which I mean the PageFlipper. We also introduced the new Animator classes, although we skimmed over some of the more complex features available to them.

This chapter is about digging deeper into Android animation, and some patterns you are likely to encounter, when using them in a real-world situation. In particular, we will look at the following:

- Using animators to parameterize more complex animations
- Applying an animation to a Fragment
- Creating an `ObjectAnimator` in XML
- Subclassing a tween animation to add extra tween animations

Let's get started!

Creating multi-variable Animators

In the previous chapter, we saw how an Animator could be applied to a view to create a simple animation, a bit like a tween. We briefly saw that there were more types of Animators available, but then I waved my hand and said something such as "don't worry about those, you will learn them once you are ready".

Today, you are ready.

As you may have guessed, one of the cool features of Animators is the ability to apply the same animation function to several different variables. There are occasions when you would want to animate two objects in parallel, so that they appear to be part of the same animation.

Using an `ObjectAnimator`, instead of creating two identical animations and applying them to different parameters, you can create a multi-variable `Animator` that controls both values at once.

Let's explore that with an example.

Time for action – making an animated Orrery

An **Orrery** is a mechanical device that models the solar system. In the middle, there is the Sun, and around it spins planets, the Moon, and occasionally fixed stars. They were used in ancient Greece to aid navigation, and they have been used as an illustrative device by the sciences since the 1700s.

They used to be made out of metal, cogs, and gears. But today we will make one out of `Drawables`, `Animators`, and `PropertyValuesHolders`. You saw that one coming, didn't you?

Enough lollygagging, let's make it! The steps for making an Orrery are as follows:

1. Create a new Android project and give it the following properties:
   - **Project name:** `Orrery`
   - **Build Target:** `Android 3.0`
   - **Application name:** `Orrery`
   - **Package name:** `com.packt.animation.orrery`
   - **Activity:** `Orrery`
2. First up, let's create the Orrery model itself. Create a new class in the project and call it `OrreryDrawable.java`
3. In `OrreryDrawable.java`, add the following code. I will interrupt it occasionally to explain the structure, but only if it is going to help us understand the animation that we will make later.

```
package com.packt.animation.orrery;

import android.graphics.Color;
```

```
import android.graphics.Point;
import android.graphics.drawable.Drawable;
import android.graphics.drawable.LayerDrawable;
import android.graphics.drawable.ShapeDrawable;
import android.graphics.drawable.shapes.OvalShape;
import android.graphics.drawable.shapes.RectShape;
import android.widget.ImageView;

public class OrreryDrawable extends LayerDrawable

{
```

4. A few facts about our orrery: You will notice that there is only one planet in this one, and that's just to keep the example simple. The only things that change with time will be the rotations of the Earth and the Moon, which we store in degrees.

```
   private static final int SPACE_HEIGHT = 150;
   private static final int RADIUS_SUN = 20;
   private static final int RADIUS_EARTH = 10;
   private static final int RADIUS_MOON = 3;
   private static final int ORBIT_EARTH = 50;
   private static final int ORBIT_MOON = 20;
   private static final int SPACE_ID = 0;
   private static final int SUN_ID = 1;
   private static final int EARTH_ID = 2;
   private static final int MOON_ID = 3;

   private float rotationEarth=0;
   private float rotationMoon=0;

   public static OrreryDrawable Create()
{
```

5. Our heavenly bodies will be defined as simple `ShapeDrawables` in the following code:

```
      ShapeDrawable space = new ShapeDrawable(new RectShape());
      space.getPaint().setColor(Color.BLACK);
      space.setIntrinsicHeight(SPACE_HEIGHT);
      space.setIntrinsicWidth(SPACE_HEIGHT);

      ShapeDrawable sun = new ShapeDrawable(new OvalShape());
      sun.getPaint().setColor(Color.YELLOW);
      sun.setIntrinsicHeight(RADIUS_SUN*2);
      sun.setIntrinsicWidth(RADIUS_SUN*2);
```

```
        ShapeDrawable earth = new ShapeDrawable(new OvalShape());
        earth.getPaint().setColor(Color.BLUE);
        earth.setIntrinsicHeight(RADIUS_EARTH*2);
        earth.setIntrinsicWidth(RADIUS_EARTH*2);

        ShapeDrawable moon = new ShapeDrawable(new OvalShape());
        moon.getPaint().setColor(Color.LTGRAY);
        moon.setIntrinsicHeight(RADIUS_MOON*2);
        moon.setIntrinsicWidth(RADIUS_MOON*2);

        Drawable[] bodies = {space, sun, earth, moon};
        OrreryDrawable myOrrery = new OrreryDrawable(bodies);

        myOrrery.setEarthPosition(0);
        myOrrery.setMoonPosition(0);
        myOrrery.setLayerInset(
               SPACE_ID,0,0,0,0);
        myOrrery.setLayerInset(
               SUN_ID,
               SPACE_HEIGHT/2-RADIUS_SUN,
               SPACE_HEIGHT/2-RADIUS_SUN,
               SPACE_HEIGHT/2-RADIUS_SUN,
               SPACE_HEIGHT/2-RADIUS_SUN);
        return myOrrery;
}

   private OrreryDrawable(Drawable[] bodies)
{
   super(bodies);
}
```

6. Let us create some new methods to calculate the positions of the heavenly bodies as they move. The things that we are interested in are the rotational position of the Earth (expressed in radians because that's how the Java `Math` class expresses angles) and the rotational position of the Moon relative to the Earth (also expressed in radians).

```
    public void setEarthPosition(float rotationEarth)
{
       this.rotationEarth = rotationEarth;
       Point earthCenter = getEarthCenter();
       setLayerInset(
       EARTH_ID,
      (int) (earthCenter.x - RADIUS_EARTH),
      (int) (earthCenter.y - RADIUS_EARTH),
```

```
                (int) (SPACE_HEIGHT - earthCenter.x - RADIUS_
EARTH),
                (int) (SPACE_HEIGHT - earthCenter.y - RADIUS_
EARTH));
```

7. The next line, shown as follows, calls `onBoundsChange` on the `LayerDrawable` to refresh the child Drawables (the Earth, Sun, and so on.)

```
      this.onBoundsChange(getBounds());
}

    public void setMoonPosition(float rotationMoon)
{
    this.rotationMoon = rotationMoon;
      Point moonCenter = getMoonCenter();
      setLayerInset(
                 MOON_ID,
                 (int) (moonCenter.x - RADIUS_MOON),
                 (int) (moonCenter.y - RADIUS_MOON),
                 (int) (SPACE_HEIGHT - moonCenter.x - RADIUS_
MOON),
                 (int) (SPACE_HEIGHT - moonCenter.y - RADIUS_
MOON));

      this.onBoundsChange(getBounds());
}

      private Point getEarthCenter()
{
      Point earthCenter = new Point();
      earthCenter.x =
      (int) (SPACE_HEIGHT/2 + ORBIT_EARTH*Math.
sin(rotationEarth));
      earthCenter.y =
      (int) (SPACE_HEIGHT/2 + ORBIT_EARTH*Math.
cos(rotationEarth));
      return earthCenter;
}

      private Point getMoonCenter()
{
      Point moonCenter = new Point();
      Point earthCenter = getEarthCenter();
      moonCenter.x =
      (int) (earthCenter.x + ORBIT_MOON*Math.sin(rotationMoon));
      moonCenter.y =
```

```
        (int) (earthCenter.y + ORBIT_MOON*Math.cos(rotationMoon));
        return moonCenter;
}

        public void setContainer(ImageView orrery)
        container = orrery;
}
```

8. We have now created two animation points within the same graphic.

9. Okay, now we need to lay it out in the GUI. Open up `res/layout/main.xml` and set it up to look as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/
android"
    android:id="@+id/main"
    android:orientation="horizontal"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
>
<ImageView
    android:id="@+id/orrery"
    android:layout_width="600dp"
    android:layout_height="fill_parent"
/>
</LinearLayout>
```

10. As you may guess from the previous code, the interesting part is the `ImageView` called `"@+id/orrery"`. This is going to be the container object for our actual orrery.

11. As you may be expecting after all these tutorials, the next thing to change will be the Activity code. Open up `Orrery.java` and add the following modules to the imports:

```
import android.animation.ObjectAnimator;
import android.animation.PropertyValuesHolder;
import android.animation.ValueAnimator;
import android.widget.ImageView;
```

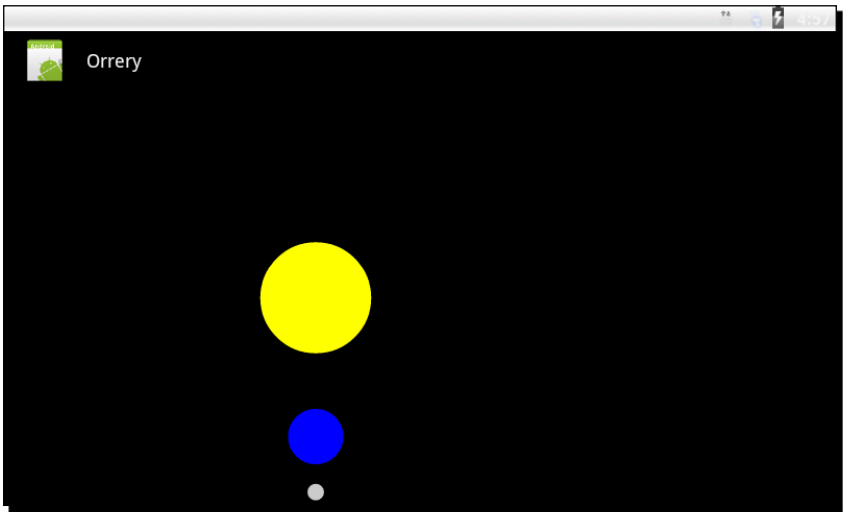
12. Here, we have added the `Animator` classes and the `ImageView` so that we can manipulate the image in `main.xml`. `PropertyValuesHolder` is a new value, and that's the point of this exercise. So let's begin and use it.

13. At the end of the `onCreate()` method, we will attach our new animation to the `ImageView`, which we declared in `main.xml`. Add the following lines:

```
ImageView orrery = (ImageView) findViewById(R.id.orrery);
    OrreryDrawable myOrreryDrawable = OrreryDrawable.Create();
    orrery.setImageDrawable(myOrreryDrawable);
```

14. This places our new animation on the screen, but we haven't added any animation properties yet.

15. Let's take a look at it anyway, so that we can see what we're working towards. Build the project and deploy it to a device or emulator. The output is shown as follows:



16. We only want to create one Animator, but we will use it to animate both the Earth and the Moon. Add the following lines:

```
PropertyValuesHolder earthPositionValues =
  PropertyValuesHolder.ofFloat(
    "EarthPosition",
    0,
    (float)(2*Math.PI));

PropertyValuesHolder moonPositionValues =
  PropertyValuesHolder.ofFloat(
    "MoonPosition",
    0,
    (float)(2*Math.PI*13));
```

17. We have our parameters. Let's add them to an `ObjectAnimator` associated with the orrery drawable, shown as follows:

```
ObjectAnimator orreryAnimator =
ObjectAnimator.ofPropertyValuesHolder(
    myOrreryDrawable,
    earthPositionValues,
    moonPositionValues);
```

18. The format is fairly simple; it's a bit like the `ObjectAnimator.ofFloat()` and `ObjectAnimator.ofInt()` that we used in the last chapter, but this time, we're adding the `PropertyValuesHolders` instead of a simple list of values. This is what allows us to apply more than one property at once.

19. Next, to do some basic housekeeping. This should look familiar from the previous chapter too. Add the following lines:

```
ValueAnimator.setFrameDelay(100);
orreryAnimator.setDuration(60000);
orreryAnimator.setInterpolator(new LinearInterpolator());
orreryAnimator.setRepeatCount(ValueAnimator.INFINITE);
orreryAnimator.setRepeatMode(ValueAnimator.RESTART);
```

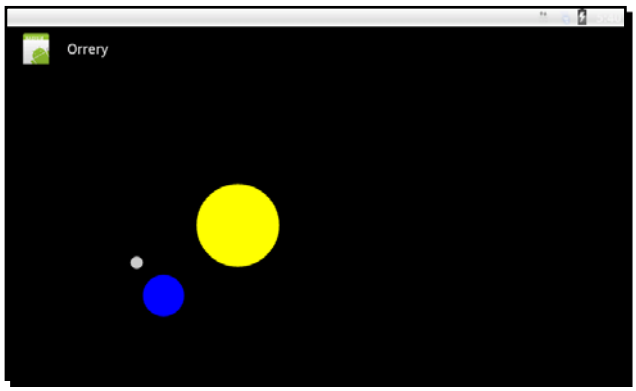
20. This should all be recognizable from earlier, but check out the following *What just happened?* section if you need a little revision.

21. Finally, right at the end of `onCreate()`, add the following line to start the animation:

```
orreryAnimator.start();
```

22. Just like the previous animators!

23. Now, our orrery is ready for use. Build and run it. The output is shown as follows:



What just happened?

Here we saw yet another approach to animation, where a single animation controls several properties. Each `PropertyValuesHolder` modified exactly one property, but the animation itself could have multiple `PropertyValuesHolders`. This is very useful when there are lots of things that you want to animate as one conceptual unit.

Please note that the simplicity of the application has meant that some of the more precise features of an orrery have had to be simplified. Please don't attempt to use this orrery for navigation; you will end up getting very, very lost!

The structure of the Orrery

When we created the Orrery, we added two animation points: `setEarthPosition` and `setMoonPosition`. They are both simple floating-point setters, but they are interdependent. That is to say, the Moon will never stay still while the Earth moves, and vice versa. Because the orrery is just a `LayerDrawable`, it does not support `tween` animations. However, we could use the `ValueAnimator` to generate a tween-like animation between two points.

Our orrery has two animated parts. The `PropertyValuesHolders` allows us to split our animation across two values. Just like when we were creating simple `ObjectAnimators` in the previous chapter, the information that we pass to the Animator is `propertyName`, (list of values). As you can see, this animation will animate the Earth rotating around the Sun (0 to 2π is a full rotation).

Also, the moon will orbit around the Earth 13 times because that's approximately how many times the real Moon orbits the Earth in a year.

Animating LayerDrawables

If you are familiar with how Android draws things, you may be wondering why we did not just call `invalidateSelf` to update the `LayerDrawable`. The truth is that `LayerDrawable` was not really intended to be used as an animation class, but we can use it as such if we are careful how we update it.

We must use `onBoundsChange` because it also updates the bounds of the `ShapeDrawables` that represent the heavenly bodies, and makes sure that the `ShapeDrawables` know about their new inset values. We will use this approach again in `setMoonPosition` too.

PropertyValuesHolder

The exact syntax for using a `PropertyValuesHolder` is as follows:

```
PropertyValuesHolder pv = PropertyValuesHolder.of[Type] (String
property, [Type] value1, value2);
```

Where `[Type]` is one of either `Int`, `Float`, `Object`, or `Keyframe`.

Helpful ValueAnimator parameters

We studied `ValueAnimators` earlier, but as a quick reminder, we will go over the parameters that we set in this example. Look at the previous code in section 10.

```
ValueAnimator.setFrameDelay(100);
orreryAnimator.setDuration(60000);
orreryAnimator.setInterpolator(new LinearInterpolator());
orreryAnimator.setRepeatCount(ValueAnimator.INFINITE);
orreryAnimator.setRepeatMode(ValueAnimator.RESTART);
```

The various terms in the previous code are explained as follows:

- The `FrameDelay` defines how frequently our Animators issue an update.
- The `Duration` is how long it will take to get from the first value in our `PropertyValuesHolder` to the last value. In this case, it will take a minute for a year to pass in our orrery.
- The interpolator is a `LinearInterpolator`, which means that it will generate a smooth and regular motion from A to B.
- The `RepeatCount` will repeat forever.
- The `RepeatMode` will be used to restart at the first point, once a cycle has been completed.

Using objects as parameters for value animations

We are now familiar with using ints and floats in animations. Let's take a look at how we would animate values of an arbitrary object.

Ints and floats are convenient because we can turn to mathematics to see what the intervening values are between one number and another. As you might guess, there will need to be some extra code to allow us to animate between two arbitrary objects.

Time for action – animating between objects

There's something not quite right about our current `OrreryDrawable` interface. For instance, you could change the motion of the Earth without ever changing the Moon. That's ridiculous! The Moon would never stay still while the Earth orbited the Sun.

There is also a potential bug in the code such as: if you update the Earth's position after updating the Moon's position, the Moon will not update itself. We clearly need a better way of accessing our position setters.

Let's change our `OrreryDrawable` so that you can only set the Earth's rotation and the Moon's rotation at the same time. This is shown in the following steps:

1. First, we will modify the `OrreryDrawable` to have the features that we want. Open up `OrreryDrawable.java` and make the two old accessors private, shown as follows:

```
	private void setEarthPosition(float rotationEarth)
{
	private void setMoonPosition(float rotationMoon)
{
```

   If you're using a Java IDE, it's probably throwing up a few errors right now. That's good, the way we are using `OrreryDrawables` is now illegal.

2. Create a new data class within `OrreryDrawable` for passing in information about the solar system. We will call it `SolarSystemData`.

```
	public static class SolarSystemData
{
	public float rotationEarth;
	public float rotationMoon;
}
```

3. We will use this data type whenever we want to parameterize the solar system from an external object, so the final thing to do is to add a setter method to `OrreryDrawable`.

```
	public void setSolarSystemData(SolarSystemData
solarSystemData) {
	setEarthPosition(solarSystemData.rotationEarth);
	setMoonPosition(solarSystemData.rotationMoon);
}
```

This is simply a bottleneck around the two old functions. You now have to set both at once.

4. We need to change the Animator code so that it uses our new `OrreryDrawable` class. We do this by adding a `TypeEvaluator` to the class that will take care of interpolating between objects that Android does not know about.

Open up `Orrery.java` and add the `TypeEvaluator` module, shown as follows:

```
import android.animation.TypeEvaluator;
```

This is just an interface; we will need to implement it.

5. So let's do that. Inside the `Orrery` class, add a private implementation of `TypeEvaluator`. The `TypeEvaluator` interface only requires one method to be implemented.

```
private class OrreryEvaluator implements TypeEvaluator
{
      public Object evaluate(
        float fraction,
        Object start,
         Object end)
{
      OrreryDrawable.SolarSystemData startSolarSystemData =
            (OrreryDrawable.SolarSystemData) start;
      OrreryDrawable.SolarSystemData endSolarSystemData =
            (OrreryDrawable.SolarSystemData) end;

      OrreryDrawable.SolarSystemData result =
            new OrreryDrawable.SolarSystemData();
      result.rotationEarth =
              startSolarSystemData.rotationEarth
            + fraction
            * ( endSolarSystemData.rotationEarth
              - startSolarSystemData.rotationEarth);
      result.rotationMoon =
              startSolarSystemData.rotationMoon
            + fraction
            * ( endSolarSystemData.rotationMoon
              - startSolarSystemData.rotationMoon);

      return result;
}
}
```

If that looks a little confusing, skip ahead to the end of this tutorial, where there is an explanation of what `TypeEvaluators` do.

6. We are going to replace our old code with the new `TypeEvaluator` code, so go into the `onCreate()` method of `Orrery.java`.

Delete the following lines, but remember where they were, because we're going to fill in their replacements where they were, shown as follows:

```
PropertyValuesHolder earthPositionValues =
PropertyValuesHolder.ofFloat(
    "EarthPosition",
    0,
    (float)(2*Math.PI));

PropertyValuesHolder moonPositionValues =
PropertyValuesHolder.ofFloat(
    "MoonPosition",
    0,
    (float)(2*Math.PI*13));

ObjectAnimator orreryAnimator =
  ObjectAnimator.ofPropertyValuesHolder(
   myOrreryDrawable,
   earthPositionValues,
   moonPositionValues);
```

Make sure you do not delete the lines where we set properties on `orreryAnimator`. Even though we're deleting the object that they relate to, we're going to replace that object with a new `orreryAnimator` that uses `OrreryDrawable.SolarSystemData` data objects.

7. In place of the `PropertyValuesHolders`, let's create two new objects that represent the start state and the end state of the solar system over a year's orbit.

```
OrreryDrawable.SolarSystemData startSolarSystemData =
  new OrreryDrawable.SolarSystemData();
startSolarSystemData.rotationEarth = 0;
startSolarSystemData.rotationMoon = 0;

OrreryDrawable.SolarSystemData endSolarSystemData =
  new OrreryDrawable.SolarSystemData();
endSolarSystemData.rotationEarth = (float) (2*Math.PI);
endSolarSystemData.rotationMoon = (float) (2*Math.PI*13);
```

8. Now, we need to create an instance of our `TypeEvaluator` so that we can use it with these objects. Add the following line:

```
OrreryEvaluator orreryEvaluator = new OrreryEvaluator();
```

Simple!

9. At last! We are now ready to put in an `ObjectAnimator` where the `PropertyValuesHolders` were. This line should come after the things we've just added, but before the lines that parameterize `orreryAnimator`, shown as follows:

```
OrreryEvaluator orreryEvaluator = new OrreryEvaluator();

OrreryDrawable.SolarSystemData startSolarSystemData = new
OrreryDrawable.SolarSystemData();
startSolarSystemData.rotationEarth = 0;
startSolarSystemData.rotationMoon = 0;

OrreryDrawable.SolarSystemData endSolarSystemData = new
OrreryDrawable.SolarSystemData();
endSolarSystemData.rotationEarth = (float) (2*Math.PI);
endSolarSystemData.rotationMoon = (float) (2*Math.PI*13);

ObjectAnimator orreryAnimator =
  ObjectAnimator.ofObject(
    myOrreryDrawable,
    "SolarSystemData",
    orreryEvaluator,
    startSolarSystemData,
    endSolarSystemData);

ValueAnimator.setFrameDelay(100);
orreryAnimator.setDuration(60000);
orreryAnimator.setInterpolator(new LinearInterpolator());
orreryAnimator.setRepeatCount(ValueAnimator.INFINITE);
orreryAnimator.setRepeatMode(ValueAnimator.RESTART);

orreryAnimator.start();
```

10. Build your application and run it! If all went according to plan, it should look exactly like the previous example did.

What just happened?

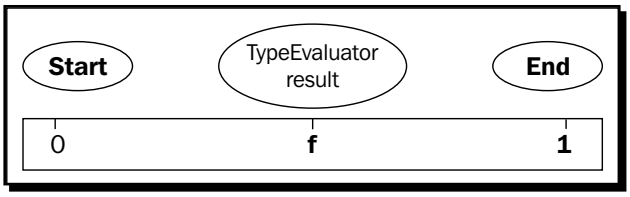
This tutorial demonstrated yet another way to use an Animator to parameterize values. This time, we saw that the "value" which it passes can be arbitrarily complex, so long as we provide a `TypeEvaluator` that can map values from zero to one to an interpolated range between two objects of the same type.

Now, the `OrreryDrawable.SolarSystemData` objects carry the exact same information that we previously represented as a pair of `PropertyValuesHolders`. Particularly, note that we can describe the relative difference in speed of the Earth and the Moon in pretty much the same way as we did before, but all of the information is now contained in one object.

This is exactly the same technique we used to animate between two numbers, but now we have a little more flexibility too.

Using a TypeEvaluator

A `TypeEvaluator` provides a simple interface between objects of a known type. It takes a start and end point and a fractional position between 0 and 1, and returns an object that equates to a point along a line where 0 = the start object and 1 = the end object. This is shown in the following image:



Typically, it means extracting some value, 'x', from each of the objects and performing the following calculation:

xresult = xstart + (fraction * (xend - xstart))

Therefore, whenever you call `typeEvaluator.evaluate(0, start, end)`, you should always get an object equivalent to `start`. Whenever you call `typeEvaluator.evaluate(1, start, end)`, you should always get back a result that is equivalent to `end`.

Setting Keyframes

We're nearly done with Animators now, but here is one last example you will find useful. Android value animations support the notion of key frames, that is, treating the state of the animation at a point as one single object.

At the start of the animation, we provide a `start` object that initializes the animation. This can be thought of as a key frame because it is one that helps define the location and behavior of the rest of the animation. Similarly, the `end` object is also a key frame, because it tells the animation where it has to reach.

Like `PropertyValuesHolders` and Animators, Keyframes support ints, floats, and objects as basic types. An example will help us understand this.

Time for action – defining fixed points with Keyframes

We said that the `Keyframe` represents a fixed point in the animation, so let's use them to represent the fixed points in the Orrery animation.

1. Open up `Orrery.java`. This is where we will make all of our changes.

   At the top of the file, add the following module:

   ```
   import android.animation.Keyframe;
   ```

   This is the standard `Keyframe` class.

2. Locate the part in the code where we declared the two `SolarSystemData` objects. We will define two keyframes based on them, shown as follows:

   ```
   OrreryDrawable.SolarSystemData startSolarSystemData =
     new OrreryDrawable.SolarSystemData();
   startSolarSystemData.rotationEarth = 0;
   startSolarSystemData.rotationMoon = 0;

   OrreryDrawable.SolarSystemData endSolarSystemData =
     new OrreryDrawable.SolarSystemData();
   endSolarSystemData.rotationEarth = (float) (2*Math.PI);
   endSolarSystemData.rotationMoon = (float) (2*Math.PI*13);

   Keyframe startFrame =
     Keyframe.ofObject(0, startSolarSystemData);
   Keyframe endFrame =
     Keyframe.ofObject(1, endSolarSystemData);
   ```

3. Underneath the new Keyframes, add a new `PropertyValuesHolder`.

   ```
   PropertyValuesHolder solarSystemFrames =
     PropertyValuesHolder.ofKeyframe(
       "SolarSystemData",
       startFrame,
       endFrame);
   solarSystemFrames.setEvaluator(orreryEvaluator);
   ```

The `PropertyValuesHolder` can be constructed with any number of frames, but we just need the `startFrame` and `endFrame` that we created earlier.

4. Finally, we need to change our `ObjectAnimator` constructor so that it uses our new Keyframe-enabled code. Find the line where we construct the `ObjectAnimator` and change it to look as follows:

```
ObjectAnimator orreryAnimator =
  ObjectAnimator.ofPropertyValuesHolder(
    myOrreryDrawable,
    solarSystemFrames);
```

   This should look familiar from the first tutorial in this chapter.

5. Build and run the activity, and make sure it's all still working in the new style.

What just happened?

We used Keyframes to define the start and end points of the animation. We didn't change any of the information; it is simply another way to describe the points of the animation.

In practice, you will have to decide whether or not Keyframes are a useful abstraction for your animation. In practice, it may be enough just to add simple floats or ints to your animation.

Using the Keyframe

In part 2 of the tutorial, we have created two `Keyframe` objects as follows:

```
Keyframe startFrame =
  Keyframe.ofObject(0, startSolarSystemData);
Keyframe endFrame =
  Keyframe.ofObject(1, endSolarSystemData);
```

The first parameter is the duration at which we want the `Keyframe` to be used. It is given as a fraction of the overall animation duration, so we put 0 for the start and 1 for the end.

The second parameter is simply the value or `PropertyValuesHolder` that defines the state of the animation at this key frame.

`ObjectAnimators` do not directly recognize `Keyframes`, we need to store them in another, different `PropertyValuesHolder`. In section 3, we also used the `PropertyValuesHolder` to pass in the `TypeEvaluator` that is required to make sense of our `SolarSystemData`.

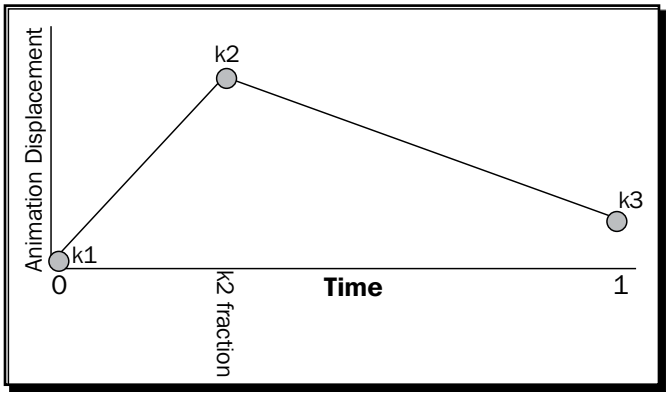
Keyframe timing

One place in which Keyframes are very useful is for scheduling portions of the animation. A `Keyframe` has a position in the animation defined by its fraction. The fraction values run from 0 to 1, so in our animation, we put `startFrame` at 0 and `endFrame` at 1.

Let's suppose we wanted to make three key frames. We could define them as follows:

```
Keyframe k1 = Keyframe.ofObject(0, k1Displacement);
Keyframe k2 = Keyframe.ofObject(k2fraction, k2Displacement);
Keyframe k3 = Keyframe.ofObject(1, k3Displacement);
```

The following graph shows how that might look in terms of animated values:



As you can see, the advantage of the `Keyframe` approach is that it lets us specify not only what value to animate but also when to get there.

The fraction is then multiplied by the duration value that is given to the animation, in order to translate the `Keyframe` time into real time that the user will see.

Pop quiz – PropertyValuesHolders, ObjectAnimators, and TypeEvaluators

1. Which of these phrases best describes `ObjectAnimator.ofObject()`?
    a. A convenient way to animate properties with simple types
    b. A way to animate multiple properties with simple types
    c. A way to animate properties with types that Android does not natively know about

2. When would it be useful to use a `PropertyValuesHolder`?
    a. When animating multiple properties at once
    b. When working with a single object of a complex type
    c When animating something with a custom interpolator
3. How does using a `PropertyValuesHolder` to list properties differ from creating multiple animators to animate the different properties?
    a. They are equivalent
    b. The `PropertyValuesHolder` animation guarantees that all animations are run at the exact same time
    c. Multiple animators are more efficient
4. When animating a list of `PropertyValuesHolder` objects, do all of the start and end values have to be equal across the different `PropertyValuesHolders`?
    a. Yes
    b. No
    c. Yes, unless you use Keyframes
5. Why would you want to define Key frames for a value animation?
    a. To control when the key frames are reached
    b. For performance reasons
    c. To use a `TypeEvaluator`

Have a go hero – tweaking the animation objects

The Moon is spinning around the Earth, the wrong way. Don't ask me how I know, just trust me. I'm a scientist.

Make the Moon spin around the Earth the correct way, by reversing its spin. You may use only the code in `Orrery.java` to achieve this.

The solution should be simple!

Combining Fragments and XML Animators

Fragments are another new feature in Android 3.0. You can think of them as Views that can be shared between applications. However, fragments can be added, moved, and removed from an application in a way that differs from ordinary views.

Changes to Fragments are applied using Transactions. Instead of setting an animation on a Fragment, you set it on the Transaction that adds or removes it from the screen.

In this example, we will also use XML for the first time in defining an Animator. It is not quite such a descriptive format as the one that the tween animations use, but I will introduce it for completeness's sake.

Enough talk, it's time to take a look.

Time for action – adding a Description Pane

As mentioned earlier, the classical orrery was a great educational piece in the natural sciences. As we are clever people, we shall add a Fragment to our orrery that provides some educational information about the solar system and we will make it fade in gracefully.

1. Firstly, let us create our animation Fragment. Add a new Java file to the project called `OrreryInfo.java`. In it, add the following code:

```
package com.packt.animation.orrery;

import android.app.Fragment;
import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.view.animation.AlphaAnimation;
import android.widget.ImageView;
import android.widget.LinearLayout;
import android.widget.TextView;

public class OrreryInfo extends Fragment
{
   @Override
   public View onCreateView(
         LayoutInflater inflater,
         ViewGroup container,
         Bundle savedInstanceState)
{
       LinearLayout result =
         new LinearLayout(getActivity());
       result.setOrientation(LinearLayout.VERTICAL);
       result.setLayoutParams(
         new LinearLayout.LayoutParams(
             LinearLayout.LayoutParams.WRAP_CONTENT,
             LinearLayout.LayoutParams.WRAP_CONTENT));
```

```
        TextView info = new TextView(getActivity());
        info.setText("Humans live on Earth, and Moon Mice live on
the Moon.  Nothing lives on the Sun, because it's a little too
hot.");
        info.setWidth(200);
        result.addView(info);

        return result;
}
}
```

This is a very simple Fragment! All we do is return a new `LinearLayout` containing a `TextView` with some interesting facts about the solar system in it.

2. Now let us create an `ObjectAnimator` to fade the new Fragment in, when we add it. Create a new XML file in `res/animator` called `fade_in.xml`. Add the following code to the new file (I'll interrupt it as we go):

```
<objectAnimator
    xmlns:android=http://schemas.android.com/apk/res/android
// The first thing we will specify is the start and end points of
the value animation.
    android:valueFrom="0"
    android:valueTo="1"
// This is how we identify the property that we want to animate.
    android:propertyName="alpha"
// And finally, as we have seen in several different sorts of
animation, the duration value!
    android:duration="2000"
/>
```

The previous concepts should all seem familiar to you by now.

3. All that remains is to add the code to the Orrery activity so that we see the new Fragment appear when we start the activity.

Open up `Orrery.java` once more and add one more module at the top of the file as follows:

```
import android.app.FragmentTransaction;
```

This is the only additional module we need to get the Fragment included in the Activity.

4. Next, we want to actually add the animation. Put the following code at the end of `onCreate()`. Firstly, we want to start a new `FragmentTransaction` as follows:

```
FragmentTransaction ft =
  getFragmentManager().beginTransaction();
```

Please note that we specify an in animation for adding the new `Fragment`, but we also add an out animation for any items that we are removing.

Our animator will be used for animators that are being added in, but in this case, we aren't removing any items, so we choose one of Android's built-in Animators just to satisfy the parameter.

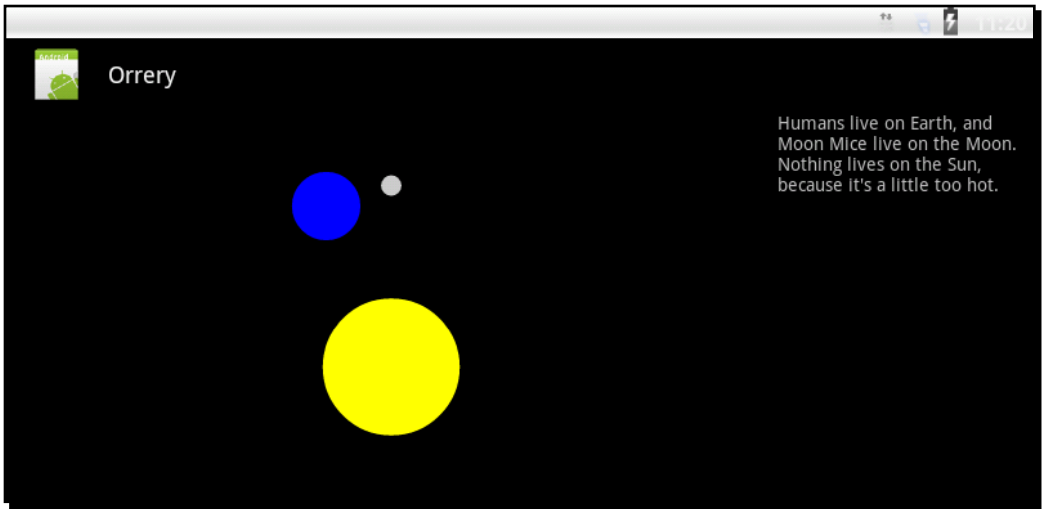
```
ft.setCustomAnimations(
  R.animator.fade_in,
  android.R.animator.fade_out);
```

Then we add a new `OrreryInfo` to the main layout, and call `commit()` to tell Android we're ready to start adding the new Fragment to the screen.

```
ft.add(R.id.main, new OrreryInfo());
ft.commit();
```

That's it! Our animated Fragment addition is complete.

5. Build and run the Activity, and observe the new information pane in action, shown as follows:



What just happened?

Two important things happened here.

Firstly, we defined an Animator in XML. This is kind of useful, but it's not such an easy format to work with as the tween animation format. However, `FragmentTransactions` do not allow us to specify a tween that was declared in Java, so we have to take this route.

Secondly, we used the Animator on a `FragmentTransaction`, so that it would be applied when our new Fragment was added to the screen.

Oh yeah, and those interesting facts about the solar system? Some of them aren't true. I'm sorry.

Declaring ObjectAnimator attributes

When you are declaring an `ObjectAnimator` in XML, the following table shows you all of the available attributes that you can give your Animator:

Attribute	Values	Description
`android:duration`	Integer milliseconds	The time taken for the animation to move from the first value to the last
`android:valueFrom` `android:valueTo`	Floats, integers, or colors in #FFFFFF format	The start and end values which the animator will animate between
`anrdoid:startDelay`	Integer milliseconds	Insert a pause for the specified time before the animation begins
`android:repeatCount`	Integer count, or INFINITE	How many times to repeat the animation. Note that this value does not include the initial playback. If you want an animation to play once, the repeat count will be 0.
`android:repeatMode`	Either "reverse" or "repeat"	For animations with a non-zero repeat count, specify whether to start the animation at the beginning, or to play the animation backwards so that it retraces its steps.

Pop quiz – Fragment animation and XML Animators

1. Where do you need to use an XML `Animator`?
    a. When you are working with animations that are not in Views
    b. When the animation uses a default interpolator
    c. When the target animation only loads Animators by resource ID
2. What class do you add an Animator to when you want to add a Fragment to a layout?
    a. The Fragment itself
    b. The Layout
    c. The `FragmentTransaction` that is adding the Fragment
3. Can you specify different animations for adding and removing Fragments with a FragmentTransaction?
    a. No
    b. Yes, you specify them separately
    c. Yes, you specify them together

Have a go hero – animating Fragments

A fade animation is a simple way to add a new `Fragment` to the scene. Another alternative would be to slide the `Fragment` into view from the right-hand side of the screen.

Change `setCustomAnimation` on the `FragmentTransaction` so that the `Fragment` slides into place on the screen.

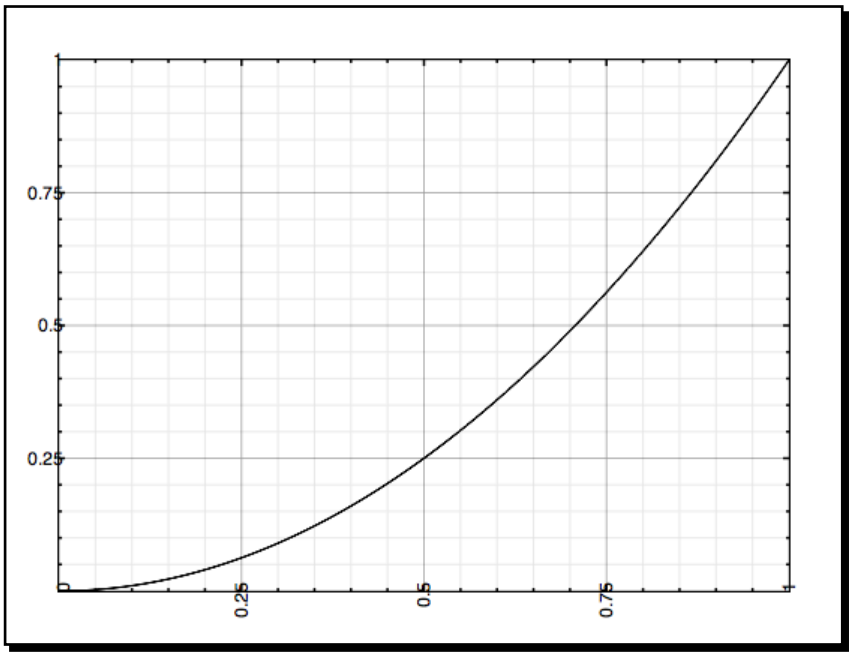
Customizing the interpolator classes

We have seen that interpolators are used in various guises in Android animations. Interpolators provide a characteristic motion to a simple animation, and it is sometimes useful to take advantage of them to provide new types of motion.

What do interpolators do?

You may have wondered to yourself how an interpolator actually works when you dig into the internals of it. Essentially, it provides a multiplier for any animation that it is associated with. It takes in a floating-point number between 0 and 1, and it gives back a floating-point number between 0 and 1.

The simplest example is a `LinearInterpolator`, which just gives back the number it was given. A more complex example is the `AccelerateInterpolator`, which gives back the square of the number, it is given. If you think about what the square of the numbers between 0 and 1 look like, this makes sense. It starts off with a gentle incline and ends up moving much faster, as shown in the following image:



The graph is of y = x2 for 0 < x < 1; you can see why it is suitable for modeling accelerating animations. If the x axis is time and the y axis is distance, you can see that the rate that distance is covered increases with the time it is accelerating.

Now, let us create our own interpolator and see what we can do with it.

Time for action – making a teleport interpolator

Our new Fragment looks kind of cool, but what would make it cooler? If it had a UFO teleport into it!

We are going to add a new UFO graphic, and we will create a new sort of interpolator to add it to the screen. The teleport looks a little bit like a flickering signal being tuned in. When it starts, it will be all flickers and no alien. But as it progresses, the flickering will become a steadier image of a UFO.

Ready? Let's go.

1. The first thing we will need is our UFO graphic. Locate the file `alien.png` from the code bundle of this chapter, and put it in your project under `res/Drawable`.

2. Next, we want to add our alien graphic to the `OrreryInfo` fragment. Open up `OrreryInfo.java` and add the following in `onCreateView`, just before the return statement as follows:

```
        ImageView alien = new ImageView(getActivity());
        alien.setImageResource(R.drawable.alien);
        result.addView(alien);
        return result;
}
```

   That has added the alien graphic to our information Fragment. Now to animate it.

3. Create a new Java file in the project called `TeleportInterpolator.java` and add the following Java code into it:

```
package com.packt.animation.orrery;

import android.view.animation.Interpolator;

public class TeleportInterpolator implements Interpolator
{
   public float getInterpolation(float input)
{
}
}
```

   This is the structure that all interpolators implement. Recall that the input will be a number between 0 and 1.

Please note that we imported `android.view.animation.Interpolator`. This is the interface that all interpolators must fit.

4. Now, we need to define the behavior that we want our interpolator to model.

   Inside the `getInterpolation()` method, add the following:

```
   public float getInterpolation(float input)
{
      return (Math.random()<input)?0.9f:0.1f;
}
```

5. Next, we add the new interpolator to an animation in the `OrreryInfo` fragment. As we will be making a teleporter, you might not be surprised to learn that we are going to apply it to an alpha animation.

Open up `OrreryInfo.java` and add in the following, before the end of the return value as follows:

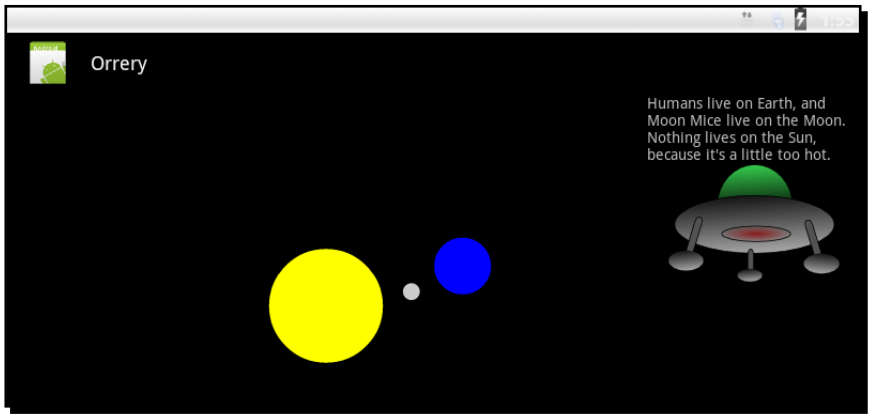
```
        ImageView alien = new ImageView(getActivity());
        alien.setImageResource(R.drawable.alien);
        result.addView(alien);

        AlphaAnimation anim = new AlphaAnimation(0,1);
        anim.setDuration(4000);
        anim.setStartOffset(4000);
        anim.setInterpolator(new TeleportInterpolator());
        alien.startAnimation(anim);
        return result;
}
```

Looks like an ordinary tween animation, right? And it is! You first encountered this kind of thing in *Chapter 3*. Here, the most interesting line is the following one:

```
        anim.setInterpolator(new TeleportInterpolator());
```

And with it we apply our new interpolator.

Give yourself a pat on the back; it's done! Now you can build and run the activity, and you will be able to see the alien UFO flicker into existence on the right, as shown in the following screenshot:



Eek! You should now see an alien!

What just happened?

We created a new class that implements an interpolator. The new interpolator has characteristics that no other interpolators offer, and you should consider creating your own interpolator subclass, when your animation needs a distinctive style of motion.

In the previous example, we made the interpolator jump randomly between something close to 0, and something close to 1. This gave it the appearance of flickering into existence. We adjusted the randomness, so that it flickers less as the animation reaches its end point. This gives the animation a feeling that it is becoming more solid with time.

Our interpolator randomly returns either 0.1 or 0.9, meaning that it is always either barely visible or nearly solid. When the animation completes, the object becomes fully solid.

It is not generally expected that an interpolator should maintain any sort of stateful information about its input; it should just convert one `float` into another.

Interpolator value ranges

Interpolators always accept a float value between 0 and 1.

It is not necessary, however, for the interpolator to return a value in the range of 0 to 1. Some interpolators actually rely on this fact to provide their animation. For instance, the `OvershootInterpolator` and `AnticipateInterpolator` go into x > 1 and x < 0 territory respectively, so that the resulting animation actually goes out of the bounds set by the start and end points.

Pop quiz – custom interpolators

1. Which of these animations can you not create your own `interpolator` for?
   a. A `ValueAnimator`
   b. A `FragmentTransaction`
   c. A `tween`
2. For which type of tween animation does it not make sense to use an interpolator that returns values outside of the domain 0-1?
   a. `AlphaAnimation`
   b. `TranslateAnimation`
   c. `RotateAnimation`

Have a go hero – modifying the Interpolator

The flickering effect is nice, but we want something that shows the UFO more clearly.

Modify the interpolator, so that the off-value (that is currently `0.1f`) increases linearly with time.

Summary

In this chapter, we uncovered some more in-depth animation features and even created an interpolator of our own!

Specifically, we covered the following:

- Using `PropertyValuesHolders` to allow us to animate more than one property on an object.
- Using a `TypeEvaluator` to allow Android Animators to treat arbitrary objects as points to animate between.
- Animating the behavior of `FragmentTransactions`.
- Creating `ObjectAnimators` in XML.
- Creating new interpolators so that our animation style can be varied.

We have now used `TypeEvaluators`, `Keyframes`, and `PropertyValuesHolders` in various combinations. You should be comfortable with using them together in any combination that seems appropriate.

Now, we have covered an awful lot about tweens, Animators, and interpolators. In the next chapter, we will use some of this knowledge to learn a handful of techniques to give us 3D graphics.

6
Using 3D Visual Techniques

In the previous chapters, we learned about some of the more common view animation techniques. In this chapter, we will re-introduce those concepts and show them how to produce a suite of visual styles around the theme of giving your animation depth.

In this chapter, we shall discuss the following:

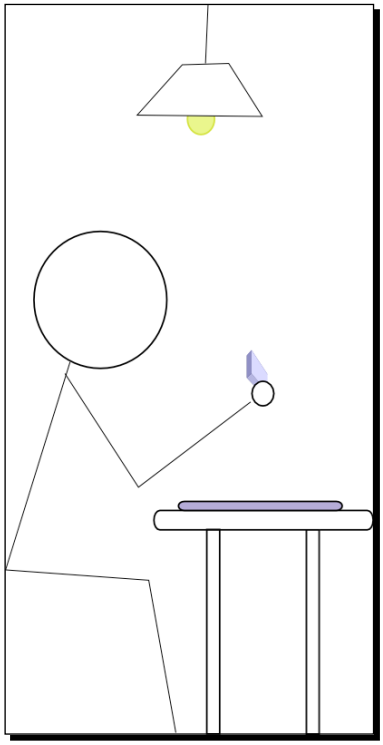
- Using animated depth effects to create a three-dimensional image
- Extending the tween animations to tween into the third dimension
- Learning a bit about 3D maths (eek!)

So let's get on with it.

Understanding 3D graphics

From lifelike video games to intuitive user interfaces, the purpose of 3D graphics is to make software more engaging to a user. As application developers, we can use that natural understanding of depth and focus to draw the user's attention and to describe the system, which is what they are working with.

Consider the following image:



To the man in the image, the piece he is holding is near to him and is in sharp focus. Everything else on the table is slightly less focused and further away, and he doesn't care quite so much about it. It's not that it's unimportant, but right now his attention is on the piece he is holding. Also, his environment is affected by the fact that there is a light source in the room, which will cast shadows when it is blocked.

To simulate that environment in an application is to suggest to our users that they are doing something natural, which they are familiar with. The 3D visual cues are helpful for understanding what is happening in your application.

In fact, simple 3D techniques are commonly used in many common application interfaces, but they are not usually animated. By animating the 3D interface, you are making the application look more natural to the user.

3D can be far more complex than this, and we'll try a more complex example later. When you want to animate something moving authentically in the third dimension, you must use some mathematical smarts to work out exactly how it should look on the screen.

I'll try not to bog you down with too many fiddly equations. You can get some neat effects by just understanding the core principles.

Showing depth with 3D effects

Depth effects are things that you can apply to any interface to make it seem more like the thing you are doing is an analogy of a real-life situation. Usually, they are simple techniques that do not require any in-depth understanding. Here, we'll take a look at three examples of 3D depth effects: raising an element, using drop shadows, and putting interesting elements in focus.

Depth effects are usually based around an interface with a fairly flat metaphor. Using the previous image as an example, the widgets in your interface are lying flat on the table. If an object contains some interesting information, the user will want to bring the object closer to their eye. Also, bringing an object closer will reinforce the idea that it is something that we are currently interested in.

Raising elements

The first thing we'll look at is perhaps the most obvious part of three-dimensional design, that is, objects appear much bigger when you bring them closer to your eye.

To help a user see that a widget is currently important and relevant, we can make it bigger and temporary.

Time for action – making a jigsaw with lifting pieces

Today, we're going to make a jigsaw puzzle. A jigsaw puzzle is similar to many interfaces that you might design in that it contains several pieces that have some interesting information on them. The user will be given a scene and will be able to swap pieces of that scene around by touching two of them to indicate that they want to swap them around.

The technique is very simple, but we are going to capture it in its own class. This way, as we add more effects to the 3D "look", they can all be kept in the same place.

As part of this example, I've used a lovely picture of my dog and me. If you want to use your own picture, you will need to split it into four pieces and resize it, so that it will fit on your tablet's screen.

If you have ImageMagick, the command to chop an image up is `convert input_file.png -crop 2x2@ +repage +adjoin output_file_%d.png`.

The steps for making a jigsaw puzzle are as follows:

1. Create a new Android project with the following settings:
   - **Project name:** `Jigsaw`
   - **Build Target:** `Android 3.0`
   - **Application name:** `Jigsaw`
   - **Package name:** `com.packt.animation.jigsaw`
   - **Activity:** `JigsawActivity`

2. Next, we want to create a new Java class that will hold a piece of the jigsaw. It will be a subclass of `FrameLayout`, so that we can add it into our main layout, and so that we can add 3D effects to it.

   Create a new Java file in your project and call it `RaisableImageView.java`. In it, add the following imports:

```
package com.packt.animation.jigsaw;

import android.content.Context;
import android.util.AttributeSet;
import android.widget.FrameLayout;
import android.graphics.drawable.Drawable;
import android.widget.ImageView;
```

   The first three imports are simply needed, so that we can subclass the `FrameLayout`. The next two will be used for interacting with our jigsaw pieces.

3. Next, we'll create the class and add a couple of constructors so that it can be used in our layout XML as follows:

```
public class RaisableImageView extends FrameLayout
{
   private ImageView image;
   private float depth=0;

   private void init(Context context,AttributeSet attrs)
{
      image = new ImageView(context, attrs);
      addView(image);
}

   public RaisableImageView(Context context, AttributeSet attrs)
{
      super(context, attrs);
      init(context, attrs);
```

```
}

   public RaisableImageView(Context context, AttributeSet attrs,
int defStyle)
{
      super(context, attrs, defStyle);
      init(context, attrs);
}
}
```

4. To calculate the third dimension, we will add a `setDepth` method to the body of the `RaisableImageView` alongside the `init()` and the two constructors as follows:

```
   public void setDepth(float depth)
{
      this.depth = depth;
      setPivotX(getWidth()/2);
      setPivotY(getHeight()/2);
      setScaleX(depth+1);
      setScaleY(depth+1);
}
```

   The `setDepth` method is a crucial part of our 3D depth effects examples. Remember it for later.

5. Finally, we add a couple of accessors to allow us to change the image in our view. This will be the way that we actually swap pieces around.

```
   public Drawable getDrawable()
{
      return image.getDrawable();
}
   public void setDrawable(Drawable drawable)
{
      image.setImageDrawable(drawable);
}
```

6. We have made a 3D-enabled class. This will become our basic view for making jigsaw pieces. Now, let us add it to a layout containing the jigsaw images.

   Import the images from the code bundle of this chapter into the `res/Drawable` directory. There should be four images. They are as follows:

```
author_dog_0.png
author_dog_1.png
author_dog_2.png
author_dog_3.png
```

7. Next, open up `res/layout/main.xml` and give it the following XML content:

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

    <com.packt.animation.jigsaw.RaisableImageView
        android:id="@+id/jigsawTopLeft"
        android:src="@drawable/author_dog_0"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />

    <com.packt.animation.jigsaw.RaisableImageView
        android:id="@+id/jigsawTopRight"
        android:src="@drawable/author_dog_1"
        android:layout_toRightOf="@+id/jigsawTopLeft"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />

    <com.packt.animation.jigsaw.RaisableImageView
        android:id="@+id/jigsawBottomLeft"
        android:src="@drawable/author_dog_2"
        android:layout_below="@+id/jigsawTopLeft"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />

    <com.packt.animation.jigsaw.RaisableImageView
        android:id="@+id/jigsawBottomRight"
        android:src="@drawable/author_dog_3"
        android:layout_toRightOf="@+id/jigsawBottomLeft"
        android:layout_below="@+id/jigsawTopRight"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
</RelativeLayout>
```

If you built and deployed the jigsaw application now, you should see a picture of a dog and a grinning maniac (or whatever picture you chose instead). Now, let's add an animated lift-and-move action.

8. Go into `JigsawActivity.java`. We first want to add a few imports to the defaults that will already be there, so let's get that out of the way.

```
import android.animation.Animator;
import android.animation.ObjectAnimator;
These will be used by the actual animation part of the jigsaw
application.
import android.view.View;
import android.graphics.drawable.Drawable;
```

These will be useful when manipulating the `RaisableImageViews`.

9. Next, we will add a list of all image IDs that we're going to work with in the jigsaw. Put this inside the body of `JigsawActivity`.

```
   private int[] pieceIDs =
{
        R.id.jigsawTopLeft,
        R.id.jigsawTopRight,
        R.id.jigsawBottomLeft,
        R.id.jigsawBottomRight
};
```

10. This game will work by letting the user pick two jigsaw pieces by tapping on them. In order to remember when the first jigsaw piece has been picked up, we store its ID in a private member variable.

```
   int firstPiece = -1;
```

11. Next, let us connect our images to an `onClick()` handler so that they can respond to touch events. Inside the `onCreate()` method in `JigsawActivity`, add the following code:

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

        for (final int pieceID : pieceIDs)
{
           View piece = findViewById(pieceID);
           piece.setOnClickListener(
          new View.OnClickListener()
{
              @Override
              public void onClick(View v)
{
```

```
                    if (firstPiece == pieceID) return;
                    v.bringToFront();
                    ObjectAnimator raiseAnimation =
                     ObjectAnimator.ofFloat(
                         v, "Depth", 0,0.3f);
                    raiseAnimation.setDuration(1000);
                    raiseAnimation.start();
                   if (firstPiece == -1) firstPiece = pieceID;
}
});
}
}
```

Right now, our application is far from being a complete game. However, if you build it now and run it, you would see that you can now click on the individual pieces and raise them all up.

12. Next, we will add a class that will swap two pieces over. Once two pieces have been "raised up" by the user, it will swap them over and lower them down again.

Because we want the image to be fully raised by the time we swap two images over, we will implement the swapping action as an `Animator.AnimatorListener` that is triggered when the raise animation is finished.

Inside the class body of `JigsawActivity.java`, add the following private class:

```
    private class PieceSwapper
     implements Animator.AnimatorListener
{
      public void onAnimationCancel(Animator animation)
{
}
      public void onAnimationRepeat(Animator animation)
{
}
      public void onAnimationStart(Animator animation)
{
}

      public void onAnimationEnd(Animator animation)
{
}
}
```

The three methods `onAnimationCancel`, `onAnimationRepeat`, and `onAnimationStart` are included simply to implement the correct interface. We are only interested in the last one: `onAnimationEnd`.

13. We want this class to operate on two `RaisableImageViews`, which we will refer to by their resource IDs. So let's add a constructor for `PieceSwapper`, where we pass this information in as follows:

```
    private int firstID, secondID;
    public PieceSwapper(int firstID, int secondID)
{
        this.firstID=firstID;
        this.secondID=secondID;
}
```

This is a fairly simple constructor, which does exactly what we need.

14. Next, let us fill out the `onAnimationEnd` method to take care of swapping the actual views to their new location, which is shown as follows:

```
    public void onAnimationEnd(Animator animation)
{
        RaisableImageView first =
            (RaisableImageView) findViewById(firstID);
        RaisableImageView second =
            (RaisableImageView) findViewById(secondID);

{
            Drawable temp = first.getDrawable();
            first.setDrawable(second.getDrawable());
            second.setDrawable(temp);
}

        ObjectAnimator dropFirst =
            ObjectAnimator.ofFloat(first, "Depth", 0.3f,0);
        dropFirst.setDuration(1000);
        dropFirst.start();

        ObjectAnimator dropSecond =
            ObjectAnimator.ofFloat(second, "Depth", 0.3f,0);
        dropSecond.setDuration(1000);
        dropSecond.start();

        firstPiece = -1;
}
```

15. Finally, we need to make sure that this new class gets called as soon as the user has selected two pieces to swap. In the `onCreate()` method, add the following lines:

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

        for (final int pieceID : pieceIDs)
{
            View piece = findViewById(pieceID);
            piece.setOnClickListener(
           new View.OnClickListener()
{

            @Override
            public void onClick(View v)
{
               if (firstPiece == pieceID) return;
               v.bringToFront();
               ObjectAnimator raiseAnimation =
                   ObjectAnimator.ofFloat(
                       v, "Depth", 0,0.3f);
                raiseAnimation.setDuration(1000);
                raiseAnimation.start();
                if (firstPiece == -1) firstPiece = pieceID;
                else
{
                  raiseAnimation.addListener(
                      new PieceSwapper(firstPiece,pieceID));
}
}
});
}
}
```

16. Your new jigsaw is ready! Compile and deploy it and have a play. It will look a little like the following screenshot:

What just happened?

Here, we used a simple technique (making things bigger) to give the impression that something is being lifted up. By scaling the images appropriately, the pieces of the jigsaw are brought into the foreground.

Let's examine some of the key methods in the previous example.

Laying out the jigsaw

Take a look at *step 2* of the example, where we set up the layout for the jigsaw. If you're familiar with the `RelativeLayout` class, you should be able to recognize what is going on. We are placing our four images on screen next to each other. The only distinguishing feature is that instead of using an `ImageView`, we are using the new `RaisableImageView`.

Also, notice the IDs that the new `RaisableImageViews` have are as follows:

```
@+id/jigsawTopLeft
@+id/jigsawTopRight
@+id/jigsawBottomLeft
```

```
@+id/jigsawBottomRight
```

Further down the example, in *step 11*, we iterate over the `pieceIDs` values that we declared as a member variable. We give each one an `onClick()` handler that raises up the image.

We added an accessor called `setDepth` to the `RaisableImageView`. An `ObjectAnimator` calls the `setDepth` accessor to handle the act of raising a jigsaw piece. We could have simply accessed the `setScaleX` and `setScaleY` properties directly, but instead we have opted to create a new abstraction for making three-dimensional changes. This will be helpful when we start adding other visual effects.

Special classes we created to help animation

We needed to create descriptive classes to describe animated actions.

The `ScalableImageView` allowed us to describe depth as a property on an image. By using this class, it became easy to animate a `ScalableImageView` getting nearer or further by changing its depth.

Scaling the image with ScalableImageView.SetDepth

In the `setDepth` method, we scale our image according to a floating-point value that will be passed in by its calling method as follows:

- If the depth is 0, this piece lies flat alongside the other pieces in our jigsaw
- If it is greater than zero, it will get bigger

For our purposes, we are less concerned about making an authentic transition to a specific depth and more interested in simply making it look like the image is being brought to the fore. For this reason, the parameters in `setDepth` are chosen to look nice rather than to be geometrically precise.

Moving pieces with PieceSwapper

We use a class called a `PieceSwapper` to swap two images over. In the `onCreate` method, you can see that we set a runnable that attaches a `PieceSwapper` to each jigsaw piece that gets chosen:

- When one piece has been picked up, we create a new `PieceSwapper`
- When two pieces have been picked up, the `PieceSwapper` will be invoked again to swap them over and lower them back to the jigsaw board

All of our animation code is called from the `PieceSwapper`.

Completing the animation with PieceSwapper.onAnimationEnd

In *step 14*, we create the `onAnimationEnd` callback that takes care of adding the jigsaw pieces to their new location. We do it by following the recipe as follows:

- Retrieve the `views` relating to `firstID` and `secondID`.
- Swap the Drawables of the two pieces that we would like to exchange. This will take advantage of the `setDrawable` and `getDrawable` methods that we added to `RaisableImageView.java`.
- Initiate a new `ObjectAnimator` for the two views to lower the two pieces back into the jigsaw board.
- Reset the `firstPiece` variable so that the user can continue playing.

Adding drop shadows

The average user is rather discerning when it comes to the third dimension. It will take more than a simple raising animation to convince them that they are interacting with the jigsaw in three dimensions. We won't stop here, but take a look at another technique for adding a 3D feeling.

Shading is perhaps the most effective technique for giving things a 3D feel. It is no accident that fast 3D shading algorithms are a big deal in the computer gaming world. Fortunately, the shading algorithms that we'll be concerning ourselves with here are considerably more humble. However, the effect that they provide can be startling.

Time for action – using shadows with our jigsaw

We want to give our jigsaw pieces a shadow when we pick them up, which will cast over the rest of the board until we put it down. We also want our shadow to raise and lower along with the animation.

Our jigsaw pieces are rectangular, which means that their shadows will also be rectangles. We will simply implement our shading technique by putting a black square behind our image, with a little bit of transparency and a slight offset between the image and the shadow.

Because we have already created a "depth" abstraction for making 3D moves, this should be easy to do in a nice smooth way.

1. Open up `RaisableImageView.java`. This is where we'll be making all of our changes. First, we need to add a few more imports to the top of the file as follows:

```
import android.graphics.Color;
import android.graphics.drawable.ShapeDrawable;
import android.graphics.drawable.shapes.RectShape;
```

   All of these new imports are going to be used for drawing rectangular shadows.

2. Moving down to the body of the class, let us add a new `ImageView` that will hold our shadow graphic. Put this in the body of the class.

```
   private ImageView shadow;
```

This will now be available throughout the class and will be manipulated in a similar way to the image itself.

3. Now, let's add our new shadow in the `init()` method. We want to create a black rectangle that will be the same size as the image itself.

```
   private void init(Context context,AttributeSet attrs)
{
      shadow = new ImageView(context,attrs);
      addView(shadow);

      image = new ImageView(context,attrs);
      addView(image);

      ShapeDrawable shadowDrawable =
          new ShapeDrawable(new RectShape());
      shadowDrawable.getPaint().setColor(Color.BLACK);
      shadowDrawable.getPaint().setAlpha(88);

      shadowDrawable.setIntrinsicWidth(
           image.getDrawable().getIntrinsicWidth());
      shadowDrawable.setIntrinsicHeight(
           image.getDrawable().getIntrinsicHeight());

      shadow.setImageDrawable(shadowDrawable);
}
```

4. We'll add some code to the `setDepth` method that indents the shadow and the image appropriately as our `RaisableImageView` gets raised.

```
   public void setDepth(float depth)
{
      this.depth = depth;
      image.setAlpha(1f);
      FrameLayout.LayoutParams imageLayout =
        (FrameLayout.LayoutParams) image.getLayoutParams();
      imageLayout.setMargins(
            (int) (getMeasuredWidth()/4*depth),
(int) (-getMeasuredHeight()/4*depth),
            (int) (-getMeasuredWidth()/4*depth),
            (int) (getMeasuredHeight()/4*depth));
```

```
        image.setLayoutParams(imageLayout);

        FrameLayout.LayoutParams shadowLayout =
           (FrameLayout.LayoutParams) shadow.getLayoutParams();
        shadowLayout.setMargins(
              (int) (-getMeasuredWidth()/4*depth),
              (int) (getMeasuredHeight()/4*depth),
              (int) (getMeasuredWidth()/4*depth),
              (int) (-getMeasuredHeight()/4*depth));
        shadow.setLayoutParams(shadowLayout);

        setPivotX(getWidth()/2);
        setPivotY(getHeight()/2);
        setScaleX(depth+1);
        setScaleY(depth+1);
}
```

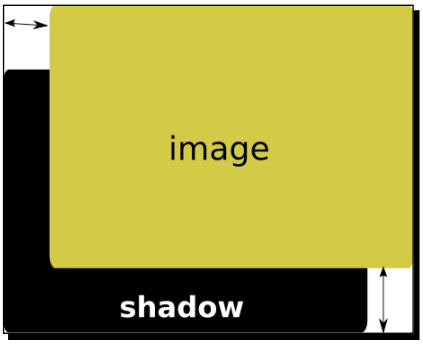
There! You have finished adding your shadow. It seems so easy now. Build your application and run it. You should now be able to see a cool shading effect, as shown in the following screenshot:

What just happened?

In a user interface, casting a shadow over a part of your design can be as simple as overlaying a dark, translucent graphic. To give the impression of a light source, indent it slightly with respect to the object that it is shadowing.

First, we added our shadow `ImageView` before the actual image. This is done so that the shadow appears before the image in the draw order, which is to say that the image will appear on top of its own shadow. We created a black, translucent `ShapeDrawable` based on the intrinsic size of the `Drawable` to hold the image that will have a shadow.

Our shadow would not be visible if it stayed behind the image all the time; we need to reveal it gradually as the jigsaw piece is raised. We do this by changing the margins that the image and the shadow adhere to when they are drawn inside the `RaisableImageView`.



The arrows in the previous image represent the left and bottom margins of the image. The `FrameLayout.LayoutParams` let us do relative adjustments to the position of the two objects in the frame during the animation.

You may be wondering how the proportions of the shadow are calculated. Again, they are only calculated to look effective, not for accuracy. The `getMeasuredHeight()` and `getMeasuredWidth()` values are used in the `setMargins` call simply, so that the proportions of the shadow are similar to the proportions of the images themselves. Use whatever values you can come up with to make your interface look pleasant.

Shadows are most effective when they contrast strongly with the background they are shadowing. Dark backgrounds do not show shadows quite so well.

Conjuring up a change in focus

One last technique to demonstrate is changing focus. When an object moves out of focus, it becomes less distinct in our vision, and it also becomes fuzzier. Android does provide a fuzzy blur effect, but it is not designed to be used with animations and can be quite slow.

More often, it is sufficient just to reduce the contrast of the thing that is moving out of focus, and that is how a lot of applications implement it. Think about how menu items look on your desktop computer, when they are out of focus. (Users of the very latest Desktop Managers may disagree with me here!).

Time for action – changing the focus of the jigsaw

At the moment, all of the pieces of our jigsaw are in focus, all of the time. If, as we brought one object to the fore, we reduced the contrast of all of the other pieces, it would enhance the visual sensation of the foreground and background.

1. Open up `RaisableImageView.java`. We are going to enhance it with a new feature: focus! Shown as follows:

```
   public void setFocus (float focus)
{
        if (depth>0) return;

        ShapeDrawable shadowDrawable =
            (ShapeDrawable)shadow.getDrawable();
        shadowDrawable.getPaint().setAlpha(255);
        image.setAlpha(focus);
}
```

2. Because `setFocus` plays around with the alpha values of our jigsaw pieces, we ought to make sure that we reset them the next time we want to use `setDepth` instead.

   At the top of `setDepth`, add the following lines:

```
        ShapeDrawable shadowDrawable = (ShapeDrawable)shadow.
getDrawable();
        shadowDrawable.getPaint().setAlpha(88);
        image.setAlpha(1f);
```

3. Now, we need to add an animation that makes use of this new focus parameter. Open up `JigsawActivity.java`, and we'll add it in here.

Firstly, we want to enhance the `View.OnClickListener` that we added to all of our jigsaw pieces. Navigate to the place in the `onCreate` method, where these are defined, and add the following lines:

```
            piece.setOnClickListener(new View.OnClickListener()
{
               public void onClick(View v)
{
               if (firstPiece == pieceID) return;

               v.bringToFront();

              ObjectAnimator raiseAnimation =
                    ObjectAnimator.ofFloat(v, "Depth", 0,0.3f);
              raiseAnimation.setDuration(1000);
              raiseAnimation.start();

              changeSiblingsFocus(pieceID,firstPiece,1,0.5f);

              if (firstPiece == -1) firstPiece = pieceID;
              else
{
                     raiseAnimation.addListener(
                       new PieceSwapper(firstPiece,pieceID));
}
}
});
```

4. Now to define that function called `changeSiblingsFocus`. In the class body of `JigsawActivity`, add the following method:

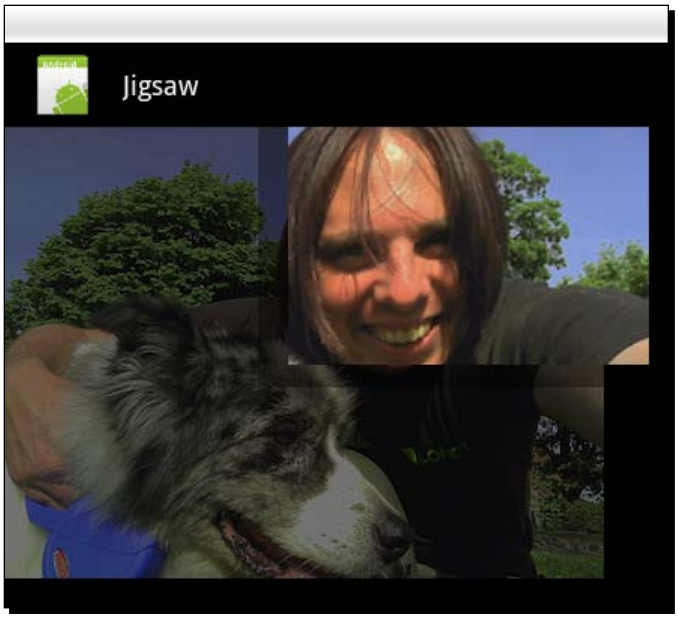
```
   private void changeSiblingsFocus(
       int callingPieceID,
       int otherPieceID,
       float fromFocus,
       float toFocus)
{
      for (int pieceID: pieceIDs)
{
         if ( callingPieceID != pieceID
           && otherPieceID   != pieceID)
{
            RaisableImageView sibling =
               (RaisableImageView) findViewById(pieceID);
            ObjectAnimator focusSibling =
```

```
                ObjectAnimator.ofFloat(
                  sibling, "Focus",fromFocus,toFocus);
            focusSibling.setDuration(1000);
            focusSibling.start();
}
}
}
```

To reset the focus of the jigsaw pieces, we need to add another call to `changeSiblingsFocus`, when we are done moving a piece. Add the following line in `JigsawActivity.PieceSwapper.onAnimationEnd`:

```
changeSiblingsFocus(firstID,secondID,0.5f,1);
```

5. Okay, you're ready to see the fruits of your work! Build and deploy the jigsaw to your device. The output will be as follows:



Notice how the raised piece stands out compared to the rest of the board.

What just happened?

Here we've taken a simple notion that the things in the foreground are more noticeable, and we've used it to guide the user's attention to the thing that they should be paying attention to. Now, the jigsaw pieces that are not in the immediate foreground get faded out, so that they do not distract the user.

Setting the image focus on a RaisableImageView

In this exercise, we introduced a new method on our `RaisableImageView` called `setFocus`. This is our new animating focus property. The following are a handful of things happening in this new method:

- The first line in the body of the method simply says, "You cannot adjust the focus at the same time as the depth". You are either in the foreground or the background, not both.
- We re-use the shadow image with full opacity.
- We fade between the image and the shadow behind it, so that moving out of focus means that the image becomes duller and less noticeable.

The `setFocus` method expects a value between 0 and 1, where 1 is completely focused and 0 is completely dark.

In `setDepth`, we first reset the alpha levels of the image and the shadows, so that they are suitable for use when changing the depth of a view. This guarantees that any image that is raised up will be at full opacity and vivid to the user.

Applying image focus to the whole jigsaw

We introduced a new method called `changeSiblingFocus` that reduces the opacity of the images that are not in focus.

The `changeSiblingFocus` method takes four arguments: the IDs of two `RaisableImageViews` that we do not want to adjust (that is, the ones that the user is interested in) and the start and end focus levels to animate between.

Every jigsaw piece that is not one of the two specified by the caller will be given an `ObjectAnimator` to animate it between the start and end values. This means that only the jigsaw pieces that are not in the foreground will get animated.

Once we had created a method to apply a focus animation to the puzzle pieces, we made a call to it when the puzzle pieces were picked up, to animate the change in focus. When the puzzle pieces were dropped at the end of a move, we made a second call to it to animate the focus being returned to normal.

We did not use a traditional blur routine because they can be slow if you need to do lots of them. We would need to add a new one for each and every animation frame we use, which would be unacceptable.

Pop quiz – depth effects

1. How can you overlay a shadow over a view?
    a. Use a `ShadowDrawable`
    b. Use a `ShapeDrawable`, the same shape as the thing it is shadowing
    c. Reduce the alpha value of the thing which is being shaded
2. What is the advantage of creating a separate parameter for describing the third dimension?
    a. It is faster
    b. It works better with tween animations
    c. It lets us change how we implement the depth effect without affecting the code which sets the depth parameter

Have a go hero – parameterizing the depth effects

All of these depth effects are parameterizable. The amount by which we raise the jigsaw piece is arbitrary, the shadow is of a particular opacity, and the un-focusing effect is done to a degree that I thought would look nice.

Try changing all of these parameters; can you find a more effective combination for giving a nice three-dimensional feel?

For added effectiveness, try specifying a different interpolator for the **focus** and **raise** effects. Are there any interpolations that look better than the default interpolator?

Creating 3D rotations

Now that you've got used to some 3D techniques, it's time to introduce some more technical features of the Android graphics APIs. In this example, we will use matrix math to rotate our jigsaw in the three-dimensional space. But don't worry! The API abstracts the actual mathematics into nice easy concepts.

Using these techniques, we essentially think of the thing we are rotating as a set of coordinates to which we apply a mathematical formula. That formula uses what we call a matrix to create the impression that the view is being moved in three dimensions.

This next example makes use of `Rotate3dAnimation` from the Android example sources. It is provided under the Apache license, and is a useful illustration of how to do 3D transformations in Android. It is also an interesting example of how to create a new subclass of tween animation.

To see the full source of this next example, visit the following site:

`http://developer.android.com/resources/samples/ApiDemos/src/com/example/android/apis/animation/Rotate3dAnimation.html`

You can view the Apache license at the following site:

`http://www.apache.org/licenses/LICENSE-2.0.html`

Time for action – spinning jigsaws

When the user first loads the jigsaw puzzle, we want the jigsaw puzzle to spin around in 3D. No real reason for this; it's just kinda cool.

We are going to use a `Camera` object, which is an object-oriented device for saying, "we want to look in this direction". In our case, the `Camera` will effectively move around the view we're animating, but this will appear to the user as if the view is spinning. I'll elaborate on Cameras later. The steps are as follows:

1. First, let us import the `Rotate3dAnimation.java` file into our project. You can get it from the following location:

   `${ANDROID_SDK_LOCATION}/samples/android-11/api/demos/src/com/example/android/apis/animation/Rotate3dAnimation.java`

   Copy this file into our jigsaw project, under the following location:

   `src/com/packt/animation/jigsaw`

2. For simplicity's sake, we are going to import the file into the same package as the rest of our Java files. Open up the `Rotate3dAnimation.java` file that you have just imported, and change the package to the following:

   ```
   com.packt.animation.jigsaw package.
   package com.packt.animation.jigsaw;
   ```

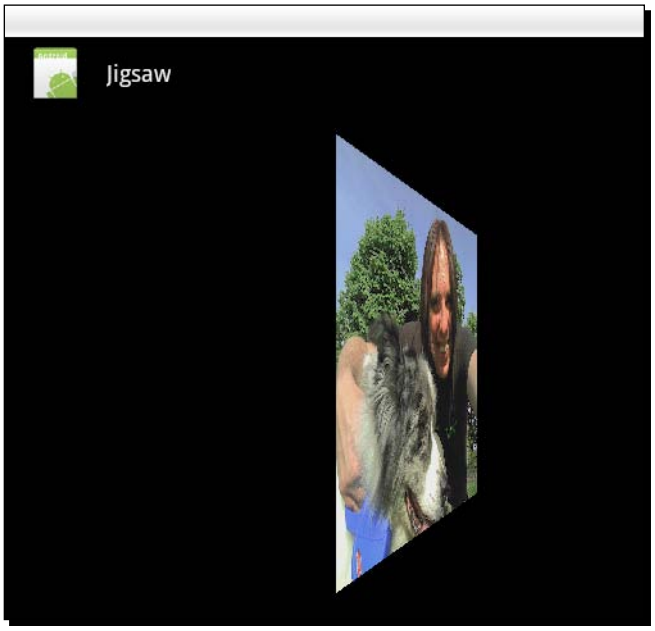
3. Now, we've got the source code for the new Animation that we're going to use, let us implement it in our `JigsawActivity`. Recall, we are going to make the whole jigsaw spin around once, when the activity starts.

   Open up `JigsawActivity.java`, and navigate to the end of the `onCreate()` method. Add the following lines:

```
View board = findViewById (R.id.jigsawbody);
Rotate3dAnimation r3d =
  new Rotate3dAnimation(0,360,300,200,0,false);
r3d.setDuration(2000);
board.startAnimation(r3d);
```

Pretty simple, yeah? It's just like any other tween animation that we might do: Construct it, parameterize, and apply it to a view.

4. Compile and run your activity, and you'll see it spin once when our activity first loads, as shown in the following image:



What just happened?

Here, we've seen that we can create new tween animation classes. We've used a `Camera`, which is a tool for creating `Matrix` objects for transforming a view in three-dimensional space. The `Matrix` did not require any special mathematics, apart from a little bit of thinking about three-dimensional space.

In fact, the `Animation` class is one of the easiest ways to apply a Matrix transformation to a view. The `Transformation` object is integral to tween animations, and it provides a simple framework for adding quite complex 3D manipulations in an animation.

Examining Rotate3DAnimation.java

The `Rotate3dAnimation` class is a tween like any other. It makes the image rotate between two given angles, in degrees. When you construct it, you say where you would like the pivot point to be, as well as the two angles to animate between.

You will see that this class extends the tween `Animation` class. We can use it anywhere in Java that we would ordinarily use a `translate`, `scale`, and so on. The first method that this class overloads is as follows:

```
    @Override
    public void initialize(
      int width,
      int height,
      int parentWidth,
      int parentHeight)
{
```

This is where the `Animation` gets some dimension information from, during initialization. In the case of `Rotate3dAnimation`, it is used to construct a `Camera` object for use later.

The other method to overload is the one that describes the actual transformation done by the animation. Here is where we do the real 3D transform.

```
    @Override
    protected void applyTransformation(
      float interpolatedTime,
      Transformation t)
{
```

It takes as its input, the output of an interpolator. Have a look at *Chapter 5* if you need to remind yourself how they work. It also takes in a `Transformation`, which is a generic way to apply the output of the animation to any view, you care to mention.

Let's take a look at the body of this method in the order as follows:

```
    protected void applyTransformation(
      float interpolatedTime,
      Transformation t)
{
        final float fromDegrees = mFromDegrees;
        float degrees =
         fromDegrees +
         ((mToDegrees - fromDegrees) *
          interpolatedTime);

        final float centerX = mCenterX;
        final float centerY = mCenterY;
```

By this point, all it does is retrieve the coordinates of the point to rotate around, and the number of degrees by which we want to rotate.

```
final Camera camera = mCamera;
```

Here, it creates a local reference to the member `mCamera` that was constructed in the `initialize()` method.

```
final Matrix matrix = t.getMatrix();
```

Recall that `t` is a generic `Transformation`. When we make changes to the `Matrix` associated with `t`, we are making changes to the view that is being animated.

```
camera.save();
```

This stores the current state of the `Camera`. In this case, the current state is the same as the default state that the `Camera` had during construction. You can retrieve the saved state by calling `restore()`, as we will see in the following few lines:

```
if(mReverse)
{
    camera.translate(0.0f, 0.0f, mDepthZ * interpolatedTime);
}
else
{
    camera.translate(0.0f, 0.0f, mDepthZ * (1.0f -
interpolatedTime));
}
```

This part of the code moves the `Camera` away from the view we are animating, as a part of the animation. We aren't going to use it for our example, but it does demonstrate another way to do 3D depth effects.

```
camera.rotateY(degrees);
```

Here, we perform a rotation about the Y axis using the `Camera`. This is the part we're interested in at the moment. It doesn't actually rotate anything yet, just updates its own internal matrix so that if you applied it to something, it would rotate it.

```
camera.getMatrix(matrix);
```

By calling `getMatrix` on the `Camera`, we get the underlying matrix (with its rotation about the Y axis). By applying this matrix to a `Transformation`, it will be applied to the view being animated.

```
camera.restore();
```

Here, we reset the `Camera` back to its saved settings. This stops us from accumulating junk data and producing nonsense results.

```
matrix.preTranslate(-centerX, -centerY);
matrix.postTranslate(centerX, centerY);
```

All rotations happen about the origin (that is, where x=0, y=0, z=0). In order to rotate about another point, we need to move the thing we are rotating, so that the point that we are rotating around becomes the origin. When we've finished our rotation, we put it back again.

Extending a tween animation

As we saw previously, there are two methods you might wish to overload when extending the `Animation` class.

When you do overload them, make sure you invoke their `super.method` as well or they will stop working.

The following are the methods that you'll probably be interested in:

initialize (int width, int height, int parentWidth, int parentHeight)

These define the dimensions of the view being animated, and the parent object that holds that view. Recall from previous tutorials (see *Chapter 3*) that you can create tweens, which take a percentage of the size of the view, as a position value for that view.

As well as setting up anything relating to dimensional translations, this method is a good place to initialize objects prior to use if you haven't already done so in the constructor.

applyTransformation (float interpolatedTime, Transformation t)

This is where things actually get animated. In a `TranslateAnimation`, it's where the view gets translated; in an `AlphaAnimation`, it's where the alpha level gets adjusted, and so on.

Describing transformations with a Matrix (android.graphics.Matrix)

If you've done any 3D graphics programming before, you will have met matrices in some form or another. In Android, unless you're using OpenGL, you only ever change matrixes by calling methods on them or by using a tool like the `Camera`.

A `Matrix` in 3D graphics is a sort of container for holding transformation data. You can use it to move things in three-dimensional space. There is all sorts of complex calculation involved in translating the human idea of "move something backwards and to the left" to something that appears to the user to move in 3D space. Fortunately, the Android APIs provide us with some nice neat methods instead.

The `Matrix` class is fairly involved, and its methods are typically described in terms of 2D space, but to give you an idea, they have names like `setTranslate()`, `setRotate()`, and `setScale()`.

You can also add other operations as pre- and post-events. Think of the code used in the previous example, where we had to do a `preTranslate()` and a `postTranslate()`, so that the center of rotation would be correct, when we applied it.

See the Android API documentation for `android.graphics.Matrix` for more information.

Doing 3D transformations with a Camera (android.graphics.Camera)

The `Camera` object provides a way to add depth to a `Matrix`. It doesn't define a new data object; it simply applies things to a `Matrix` in a three-dimensional way. You will need to use both of these classes in order to do a 3D transform.

Typically, the way you use the `Camera` is like we did in the previous example. You use the `Camera` to specify your 3D rotation, and then you call `getMatrix()`, which gives you a `Matrix` that provides the transformation you requested. Once you have that `Matrix`, you can prepend or append more transformations to it.

The following are a few methods from the `Camera` object that you may find useful:

rotateX (float), rotateY (float), rotateZ (float)

These create a `Matrix` that rotates points about the X, Y, or Z axis. In the previous example, we rotated the jigsaw right around the Y axis.

translate (float x, float y, float z)

This does a proper 3D depth transformation, including a Z axis.

save() and restore()

These two methods refer to the `Camera` state. Because you can make incremental translations and rotations, your code might get confusing if you have to undo some part of the `Camera` transformation. This way, you can save a waypoint in your transformations and get back to it by calling `restore()`.

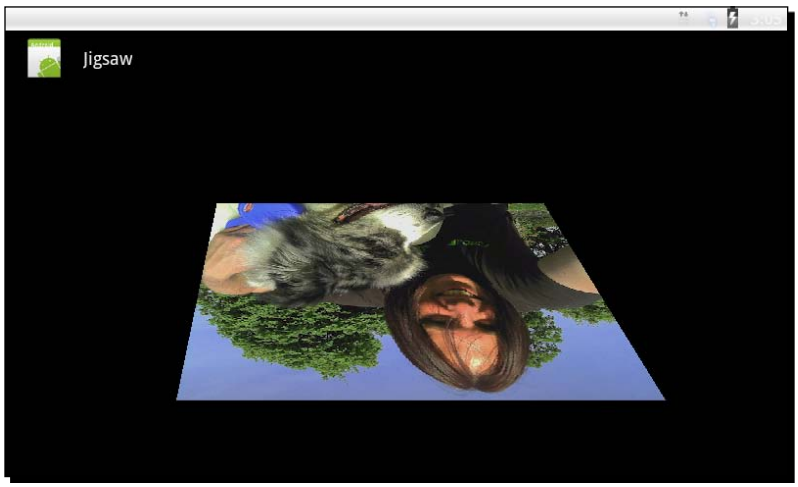
Pop quiz – 3D rotations

1. What is the `Transformation` class for?
   a. Manipulating `Matrixes`
   b. Applying a matrix transformation to an arbitrary view
   c. Interpolating between two values
2. Where might you use `android.graphics.Camera`?
   a. To create 3D matrix transforms
   b. To save a view to disk
   c. To take pictures
3. Which methods would you be likely to overload if you were subclassing a tween animation? (Pick 2)
   a. `applyTransformation`
   b. `initialize`
   c. `onDraw`
   d. `onResize`

Have a go hero – rotating along a different axis

The `Rotate3dAnimation` is limited to rotations about the Y axis. However, it would be better if it rotated about the X axis instead.

Modify `Rotate3dAnimation` so that the rotation looks like the following image:



Identify the axis about which this image is rotating, and try to duplicate it.

Summary

In this chapter, we covered a lot of things about animating in the third dimension.

Specifically, we covered the following:

- How to add emphasis to a view by making it appear to be brought closer to the user
- Adding shadows to enhance the appearance of depth
- Reducing the opacity of a view to make it appear to go out of focus
- Three-dimensional transformations in an animation

In the first half of this chapter, we did not use any "true" 3D techniques. We simply used visual cues, which appear to the user as if they are 3D.

In the second half of this chapter, we used true 3D techniques to animate a view being rotated about a point.

We also saw how to subclass a tween animation to apply a tween effect, which was previously unavailable.

We're done with 3D for now. Let's move on to the next chapter, where we will look at animating big scenes based on a looping control function.

7
2D Graphics with Surfaces

We've seen that Android provides a wealth of features for creating animations by interchanging and manipulating images. But what if we want direct control of the code that performs the animation?

By combining an Android display element called a **surface***, with a technique from computer game programming called a* **game loop***, we can programmatically create animations in an efficient way. This is useful for creating fast animated widgets with a unique style.*

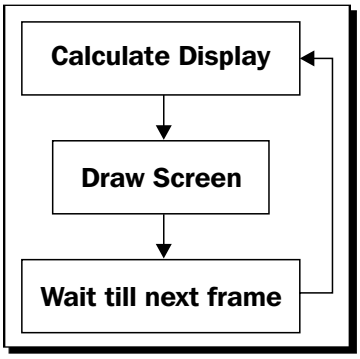
In this chapter, we shall do the following:

- Create a surface and make it usable
- Write a game loop
- Learn about some drawing tools that are useful for animation
- Optimize our animation for smoothness

So let's get on with it.

Introducing game loops

Game loops are single-threaded routines that are responsible for everything associated with the "game" that they implement. They control graphics, input, and computational updates. The structure of the game loop is pictured as follows:



Whenever we talk about game loops, this is the idea that we will be referring to. It may gain extra methods or tasks, but the things we need to do are always the same; decide what to draw on the screen, and then draw it. Repeat this loop until the animation ends.

I will talk about the different parts of the game loop throughout this chapter, so keep this flowchart in mind! Whenever you want to write a display loop that manages its own updates in an application, you will use a game loop pattern like this, or a variation. The last example in this chapter contains an example of a variation in a game loop.

Drawing a surface on the screen

A surface is essentially a rectangle of screen space, such as a blank sheet of paper without any view management code or anything like that. You can write graphics to a surface via a `Canvas` object; there are other tools to address a surface, but the `Canvas` is the most interesting one for us right now because it gives us some good tools for drawing our animation.

We now know a little bit about surfaces, but how do we intend to draw one on the screen? There is a convenient view subclass called `SurfaceView` that takes care of the boundary between our raw surface area and any Android views that may be on the screen. In the next example, we will use a `SurfaceView` to draw some bubbles, and a game loop to animate it.

Time for action – animating bubbles on a surface

A popular psychologist has discovered that the most relaxing thing in the world is watching bubbles rise up from the bottom of a glass. Previously, people thought that it was the liquid contents of the glass that were relaxing, but now science has told us that it is the bubbles that give us that soothing feeling.

Well we've had a busy day writing Android apps, and now it's time to unwind by watching some nice relaxing bubbles. But first we have to write another Android application to see them! Well, writing Android apps is kind of relaxing.

In this Activity, we will subclass `SurfaceView` to get us a direct surface onto the screen, and then create a game loop thread that will draw animated bubbles.

This will be a much more technical animation technique than the ones we have seen previously; it will be an eye-opening experience! The steps for this animation technique are as follows:

1. Create a new Android project, and give it the following properties:
   - **Project name:** `Bubbles`
   - **Build Target:** `Android 3.0`
   - **Application name:** `Bubbles`
   - **Package name:** `com.packt.animation.bubbles`
   - **Activity:** `BubblesActivity`

2. We will need an object to describe a bubble, so that we can draw it on screen. This class will be responsible for retaining its position and velocity on screen, and also for drawing itself.

   Create a new class called `Bubble.java` with the following content:

```
package com.packt.animation.bubbles;

import android.graphics.Canvas;
import android.graphics.Color;
import android.graphics.Paint;

class Bubble
{
   private float x, y, speed;
   private static final Paint bubblePaint = new Paint();
   static
{
      bubblePaint.setColor(Color.CYAN);
```

```
        bubblePaint.setStyle(Paint.Style.STROKE);
    }

    public static final int RADIUS = 10;
    public static final int MAX_SPEED = 10;
    public static final int MIN_SPEED = 1;

    public Bubble (float x, float y, float speed)
    {
        this.x = x;
        this.y = y;
        this.speed = Math.max(speed, MIN_SPEED);
    }

    public void draw(Canvas c)
    {
        c.drawCircle(x, y, RADIUS, bubblePaint);
    }

    public void move()
    {
        y -= speed;
    }

    public boolean outOfRange()
    {
        return (y+RADIUS < 0);
    }
    }
```

3. Next, we need to create a surface to hold all of this information. Create a new class called `BubblesView.java`, which subclasses `android.view.SurfaceView`.

```
package com.packt.animation.bubbles;

import android.content.Context;
import android.util.AttributeSet;
import android.view.SurfaceView;

public class BubblesView extends SurfaceView
{
    private LinkedList<Bubble> bubbles = new LinkedList<Bubble>();

    public BubblesView(Context context, AttributeSet attrs)
    {
```

```
        super(context, attrs);
}
}
```

This is sufficient for Android to recognize our object as a view, so let's add it to our application's layout. It won't do anything useful just yet!

4. Open up `res/layout/main.xml` and add our new `BubblesView` as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/
android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
>

<com.packt.animation.bubbles.BubblesView
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
/>
</LinearLayout>
```

5. Open up `BubblesView.java` again, we will add a little more to it.

   At the top of the class, add the following import and add it as an interface on `BubblesView`:

```
import android.view.SurfaceHolder;

public class BubblesView
    extends SurfaceView
    implements SurfaceHolder.Callback
{
```

6. Now implement the methods prescribed by the `SurfaceHolder.Callback` interface. In order to retrieve the `SurfaceHolder` to draw on, create a private member in `BubblesView` to hold it as follows:

```
   private SurfaceHolder surfaceHolder;
```

   Add the following methods to `BubblesView` to complete the implementation. Currently, we are only interested in `surfaceCreated`, which is shown as follows:

```
   public void surfaceChanged(
      SurfaceHolder holder,
       int format,
      int width,
```

```
        int height)
{
}

    public void surfaceCreated(SurfaceHolder holder)
{
        surfaceHolder = holder;
}

    public void surfaceDestroyed(SurfaceHolder holder)
{
}
```

7. We need to associate our `SurfaceHolder.Callback` methods with the `SurfaceView` that we are implementing. In the constructor for `BubblesView`, add the following line:

```
    public BubblesView(Context context, AttributeSet attrs)
{
        super(context, attrs);
        getHolder().addCallback(this);
```

To be able to draw on the surface, we need to implement `SurfaceHolder.Callback` and pass our implementation to `getHolder().addCallback`. If we don't do this, we will not receive the `SurfaceHolder` that we want to draw on.

8. Next, let's implement the `drawScreen` portion of the game loop. Include the following new import statements at the top of `BubblesView.java`:

```
import java.util.LinkedList;
import android.graphics.Canvas;
```

Add a private member variable in the body of `BubblesView` as follows:

```
    private LinkedList<Bubble> bubbles = new LinkedList<Bubble>();
```

Then create a private method like the following:

```
    private void drawScreen(Canvas c)
{
        for (Bubble bubble : bubbles)
{
            bubble.draw(c);
}
}
```

9. Next, we shall add another method to handle the `calculateDisplay` portion of the game loop as follows:

```
   private void calculateDisplay(Canvas c)
{
      randomlyAddBubbles(c.getWidth(),c.getHeight());
      LinkedList<Bubble> bubblesToRemove = new
LinkedList<Bubble>();
      for (Bubble bubble : bubbles)
{
         bubble.move();
         if (bubble.outOfRange())
         bubblesToRemove.add(bubble);
}
      for (Bubble bubble : bubblesToRemove)
{
         bubbles.remove(bubble);
}
}
```

We also need to say how we are going to add bubbles. Add a private constant `BUBBLE_FREQUENCY` to `BubblesView`, such as the following:

```
   private float BUBBLE_FREQUENCY = 0.3f;
```

Now implement the `randomlyAddBubbles` method in `BubblesView` as follows:

```
   public void randomlyAddBubbles(
      int screenWidth,
      int screenHeight)
{
      if (Math.random()>BUBBLE_FREQUENCY) return;
      bubbles.add(
            new Bubble(
                  (int) (screenWidth*Math.random()),
                  screenHeight+Bubble.RADIUS,
                  (int) (Bubble.MAX_SPEED*Math.random())));
}
```

This method randomly adds a bubble whenever `Math.random()>BUBBLE_FREQUENCY`, at the bottom of the screen but with a random X position.

10. Now we have a surface to draw on and we have a `Bubble` class to perform display calculations. We now need to implement a game loop to tie them together as an animation.

Create a private `Thread` class in `BubblesView` called `GameLoop`. Its `run()` method will be the game loop. I have highlighted the lines that represent the game loop parts as follows:

```
   private class GameLoop extends Thread
{
      private long msPerFrame = 1000/25;
      public boolean running = true;
      long frameTime = 0;

      public void run()
{
         Canvas canvas = null;
         frameTime = System.currentTimeMillis();
         final SurfaceHolder surfaceHolder =
            BubblesView.this.surfaceHolder;
         while (running)
{
            try
{
            canvas = surfaceHolder.lockCanvas();
            synchronized (surfaceHolder)
{
       calculateDisplay(canvas);
       drawScreen(canvas);
}
}
finally
{
            if (canvas != null)
            surfaceHolder.unlockCanvasAndPost(canvas);
}
       waitTillNextFrame();
}
}
}
```

11. Here it is! Add this method to the `GameLoop` class as follows:

```
      private void waitTillNextFrame()
{
```

```
            long nextSleep = 0;
            frameTime += msPerFrame;
            nextSleep = frameTime - System.currentTimeMillis();
            if (nextSleep > 0)
{
               try
{
                     sleep(nextSleep);
}
catch (InterruptedException e)
{
}
}
}
```

This method calculates when the next frame is due to be drawn, then waits for that length of time.

Self-optimizing game loop

If our game loop starts taking too long, then the `nextSleep` value will go negative. This is very unlikely in our application, but if you are doing something that takes a lot of processor power, you may want to detect when `nextSleep < 0` and remove non-essential computations.

12. Next, we need to associate this new game loop with our application. We will do this by starting it as soon as Android gives us a `SurfaceHolder`. First, add our game loop as a member of `BubblesView` as follows:

```
private GameLoop gameLoop;
```

We want to create it afresh when the surface is created, and destroy it when the surface is destroyed. We shall rewrite our `surfaceCreated` and `surfaceDestroyed` methods to use two new methods called `startAnimation` and `stopAnimation` respectively as follows:

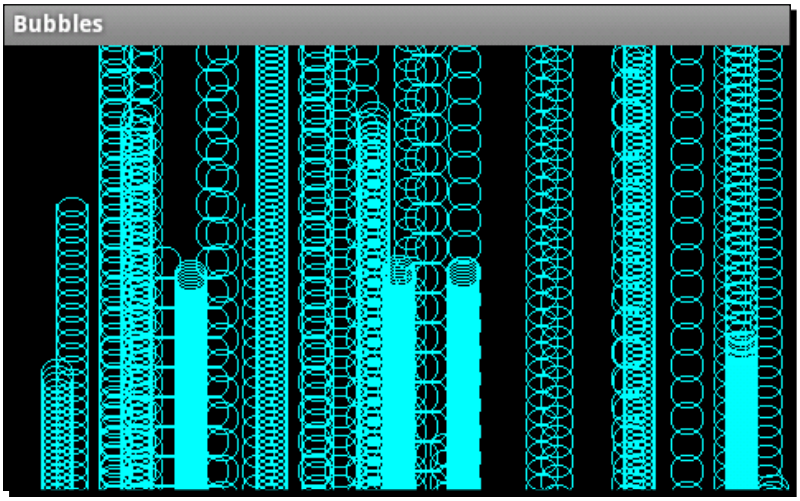
```
@Override
public void surfaceCreated(SurfaceHolder holder)
{
    surfaceHolder = holder;
    startAnimation();
}

@Override
public void surfaceDestroyed(SurfaceHolder holder)
```

```
{
      stopAnimation();
}

   public void startAnimation()
{
      synchronized (this)
{
         if (gameLoop == null)
{
            gameLoop = new GameLoop();
            gameLoop.start();
}
}
}

   public void stopAnimation()
{
      synchronized (this)
{
         boolean retry = true;
         if (gameLoop != null)
{
            gameLoop.running = false;
            while (retry)
{
               try
{
                  gameLoop.join();
                  retry = false;
}
catch (InterruptedException e)
{
}
}
}
         gameLoop = null;
}
}
```

The highlighted new code ensures that the existence of a game loop is strictly linked to the presence of a valid `SurfaceHolder`.

Always be careful to ensure that the game loop is halted when the `surfaceDestroyed()` method is called, or your Activity will be in an unstable state.

13. Build and run the application, and you will see something similar to the following image:



14. We want to clear the whole screen at the start of each frame. We will then redraw the bubbles one-by-one as we do currently.

To do this, import a couple more classes at the top of `BubblesView.java` as follows:

```
import android.graphics.Color;
import android.graphics.Paint;
```

We are going to use these new classes to clear the screen in blue, every time we draw a frame. Inside the body of the class, add a new member variable as follows:

```
   private Paint backgroundPaint = new Paint();
```

In the constructor for `BubblesView`, set the color for the paint, such as the following:

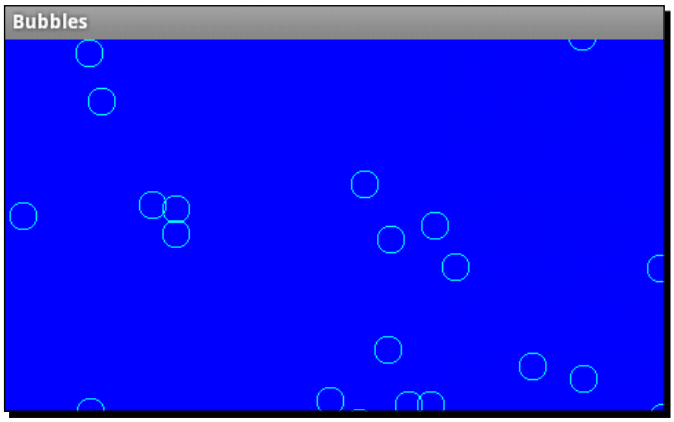
```
   public BubblesView(Context context, AttributeSet attrs)
{
      super(context, attrs);
      getHolder().addCallback(this);
      backgroundPaint.setColor(Color.BLUE);
}
```

Lastly, in `drawScreen`, fill the screen with a blue rectangle every time the frame is redrawn as follows:

```
   private void drawScreen(Canvas c)
{
      c.drawRect(
         0,
         0,
         c.getWidth(),
         c.getHeight(),
         backgroundPaint);

      for (Bubble bubble : bubbles)
{
         bubble.draw(c);
}
}
```

This will get rid of any graphics that are already on the surface, when we begin the drawing routine.

15. The proof of the bubbles are in the bubbling! Build and run `BubblesActivity` and have a look at the following screenshot:



As you can see, the bubbles are much more authentic now!

What just happened?

We used a `SurfaceView` to make a programmatic animation based on a simple routine. We implemented a game loop to update the states of our bubble objects and to draw on the surface via a `Canvas`.

As we saw, the surface is very simple and needs a lot of housekeeping. Surface-based graphical objects don't handle their own drawing like views do; that work is entirely up to us and we must be careful to redraw the screen once it has been changed. When we tested the animation at stage 12, we saw that it was full of bubbles that weren't being erased after they had been drawn.

We also saw that there are several components to a surface-based animation. We need the following:

- A game loop to implement all of the visible animation code
- A `SurfaceView` to embed our graphics in a layout
- A `SurfaceHolder` to allow us to access the surface, via a `Canvas` object
- A `SurfaceHolder.Callback` to get a hold of the surface, when it is created

All of these elements are essential parts of a surface-based animation. The apparent complexity is due to the fact that we are abandoning all of the convenience methods that an ordinary views-based GUI would provide. It is for this reason that you should consider very carefully before deciding that your animation would be best implemented by using a `SurfaceView`.

The design of the Bubbles application

There is a lot of code in the previous example that is needed to make an animation, and it does not follow a fixed format in the same way that other animations do. A tween animation, for instance, just needs a set of component parts to be joined together like a daisy chain, but so far the only structure that we have seen in the Bubbles example is that it uses a game loop.

Let's take a look at how the Bubbles application fits together. You don't have to use this structure in your own `SurfaceView` animations, but you might want to consider something similar.

Investigating Bubble.java

The key methods in this class are as follows:

- The constructor: This provides the initial values for position and speed of the bubble (all bubbles move upwards, that's what bubbles do).
- `draw(Canvas c)`: This draws the bubble onto the screen. This will be called in the `drawScreen` part of our game loop.
- `move()`: This moves the bubble upwards a little. Our game loop will call this for each bubble during the `calculateDisplay` phase.

- `outOfRange()`: This becomes true as soon as the bubble leaves the top of the screen. This will be called during `calculateDisplay`, after the `move()` calculations have been made.

It is significant that `Bubble.java` does not inherit any other classes.

Previously, all of our animation classes have been subclasses of existing view classes so that Android could draw them on the screen. Now we are going to be responsible for that task!

Investigating BubblesView.java

This class implements our `SurfaceView`.

A `SurfaceView` does not immediately know where it can draw its data to, but it receives a `SurfaceHolder` object at runtime, once the Android device has allocated some screen space.

In order to access the `SurfaceHolder`, we must attach a `SurfaceHolder.Callback` object to the `SurfaceView` object. This means that we can receive a surface contained in a `SurfaceHolder` and subsequently write to the display.

The `BubblesView` contains a `LinkedList` of Bubble objects for it to draw whenever the `drawScreen` part of the game loop happens.

Game loop

`BubblesView` also contains the game loop itself as a nested class definition. Each part of the game loop is given its own method in `BubblesView.java`.

Please note that the `Canvas` needs to be acquired by calling `surfaceHolder.lockCanvas()` and released by calling `surfaceHolder.unlockCanvasAndPost(canvas)`. This lets Android know when you're working on the surface, and it won't try to draw a half-finished screen or try to delete memory you are working with. For similar reasons, these calls are wrapped in a synchronized block.

The `msPerFrame` and `frameTime` values will be used in the `waitTillNextFrame` method.

calculateDisplay

This method completely describes how bubbles are modeled. Given a screen, we can do the following activities in `calculateDisplay`:

- Randomly add bubbles
- Move the bubbles up the screen
- Remove bubbles once they go out of range at the top of the screen

Seeing the game loop in action

As you saw, we implemented the game loop as its own thread. Unlike views, which are called from Android's GUI thread, we have to explicitly request access to the `Canvas` object that we are going to draw on to.

Notice that in step 10, the parts of the loop that depend on a `Canvas` must be wrapped in a try-finally loop that retrieves the actual `Canvas`. See `SurfaceHolder`, which is given as follows, for more information about why that is necessary.

The game loop in this example contains code, which ensures that the animation updates a particular number of times every second. You will be familiar with that idea from things such as `ValueAnimators`. At the end of this chapter, you will see that this isn't absolutely necessary, and we will learn another approach for updating the screen.

It is generally expected that an application that uses a game loop is single-threaded. Game loops are ideal if you want to avoid the overheads of an event-driven multi-threaded application. Of course, you can't avoid multithreading entirely in Android, but you can reduce the overhead dramatically. For instance, in the previous tutorial, we can add and remove things from a `LinkedList` without having to wrap it in synchronized blocks. Because our code is single-threaded, we know that we will never need to protect against simultaneous reading and writing of our data members.

A cautionary tale about surfaces

When I first found out about surfaces, I thought that you could update them incrementally and it would still work, that is, you only need to redraw the parts that are being updated. Unfortunately, Android does not guarantee that this will work.

When drawing a surface, Android typically switches between two copies of the screen, known as buffers. This is so that you can lock one buffer and draw your animation on it, while the other buffer is being drawn to the screen. When you unlock the `Canvas`, Android swaps the buffers over. But the next time you draw on the screen, it won't contain the things you just drew!

The moral of the story is this: always redraw everything in your `Canvas`, every frame. Otherwise your display might get corrupted.

Using a SurfaceView

The `SurfaceView` provides the link between the Android graphic's APIs and the raw screen access that a surface provides.

In the example we just made, we did not strictly need to override any of the methods in `SurfaceView`, but it is a common pattern to subclass `SurfaceView` anyway in this situation, as it provides a neat encapsulated object that can be drawn as part of a layout.

The `SurfaceView` does not have an underlying `Surface` object when it is created. Because of this, we do not draw onto the `SurfaceView` until we have been given a `SurfaceHolder` through `SurfaceHolder.Callback`.

Using a SurfaceHolder

The `SurfaceHolder` is just a container class for getting a surface, a block of screen space that we can draw our animation on.

You can acquire a raw `Surface` object directly, but they are not immediately useful. Instead, we get a `Canvas` object.

lockCanvas

Because the underlying surface is not constrained by the views system, it is volatile and a lot of things could go wrong if you were trying to access the `Canvas` at the same time as another thread. That is why we call `lockCanvas()` to get access to a `Canvas` that wraps the surface. It means that we have safe control of the `Canvas` until we explicitly release it.

unlockCanvasAndPost

This is the twin function of `lockCanvas`, which releases our ownership of the `Canvas`. It is very important that we do this, or Android will not be able to redraw that area of screen. It is for that reason that we wrap the call in a try-finally block, to ensure that both calls are made.

Recall from the previous example:

```
            while (running)
    {
                try
    {
                    canvas = surfaceHolder.lockCanvas();
                    synchronized (surfaceHolder)
    {
                       calculateDisplay(canvas);
                       drawScreen(canvas);
    }
```

```
    }
    finally
    {
                    if (canvas != null)
                    surfaceHolder.unlockCanvasAndPost(canvas);
    }
                waitTillNextFrame();
    }
```

Unlocking the `Canvas` works a bit like double-buffering in other graphical systems; the screen will not be redrawn until the surface is declared ready for use by calling `unlockCanvasAndPost`. But `unlockCanvasAndPost` is not exactly like double-buffering; read the cautionary tale about surfaces, mentioned previously, for more information.

Double-buffering is a term from computer graphics. When you double-buffer an animation, it means that you work on a "back buffer" that does not get drawn to the screen. When you have finished working on it, you copy the whole buffer to the screen in one go.

Using a SurfaceHolder.Callback

The `SurfaceHolder.Callback` is used by Android to announce that the screen is ready to be drawn on by our application. It provides three callback methods that we can use to tell when a screen can be drawn on.

surfaceCreated (SurfaceHolder holder)

This method is called as soon as it is safe for our application to write to the screen.

The `SurfaceHolder` that is passed in can be used for all of our graphical updates. For instance, in our previous example, we store it for use within the game loop.

```
    public void surfaceCreated(SurfaceHolder holder)
{
        surfaceHolder = holder;
        gameLoop = new GameLoop();
        gameLoop.start();
}
```

We also started the game loop at this point. There is no point starting the game loop until there is something for us to draw!

This method may be called more than once during the lifecycle of your application. For instance, if the application is paused and resumed, you might find that the screen space needs to be reallocated. For this reason, be prepared to stop and start your application's game loop appropriately.

surfaceDestroyed(SurfaceHolder holder)

This is the twin of the `surfaceCreated` method. Once this method has been received, you should not use any surface that was previously in use. In our code, we stop the game loop, so that there are no further graphical changes made.

```
   public void surfaceDestroyed(SurfaceHolder holder)
{
      boolean retry = true;
      gameLoop.running = false;
      while (retry)
{
         try
{
            gameLoop.join();
            retry = false;
}
catch (InterruptedException e)
{
}
}
}
```

Please note how strictly we apply the rule that the game loop must stop. If any writes did occur to a deleted surface, our application would force close immediately.

surfaceChanged(SurfaceHolder holder, int format, int width, int height)

This method is called if the surface undergoes any structural changes, such as if it changes size or starts using a different pixel format.

Because we were accessing all information like dimensions via a `Canvas` wrapper, this took care of all of this housekeeping work for us in this instance. If, however, you need to store things like width and height, or you need to know what the pixel format is, then this method is for you.

If you aren't sure what a pixel format is, feel grateful! And keep using the `Canvas` wrapper for doing graphical changes.

Using the Canvas as an animation tool

In the previous example we saw that the `Canvas` representation of a surface allows us to draw hollow circle shapes and filled rectangles onto the surface.

The `Canvas` is a generalized drawing interface, and provides a whole suite of tools for drawing things onto its internal representation. Changing the `Paint` object that we pass to them can further extend the Canvas' drawing methods.

In the next example, we will take a look at a few more of the effects that can be used, and we will finish up by making the bubbles scene look a bit more realistic!

Time for action – making more realistic bubbles

Our bubbles scene needs a more realistic bubble. At the moment it looks a little bit like it was written for a computer in the 1980s.

Let us explore the `Canvas` and `Paint` tools to find a more effective graphical representation of a bubble. As we experiment with the available tools, we'll learn more about how to apply `Canvas` graphics to the display.

1. Hey, I've got an idea! They say a picture is worth a thousand words, well what if they're wrong? Let's try replacing the picture of the bubble with the word "bubble".

   Good idea, eh? Open up `Bubble.java` and navigate to the `draw()` method. Replace the call to `drawCircle` with one to `drawText`, like the following:

```
   public void draw(Canvas c)
{
      c.drawText("Bubble", x, y, bubblePaint);
}
```

   This makes use of a different `Canvas` drawing tool, `drawText`.

2. Okay, let's see how it looks. Build and run the application. The following screenshot will be displayed:

   Okay, now I've seen it, I don't mind saying that was a pretty dumb idea of mine. On the positive side, at least we've seen that we can make animated, moving text on a surface.

3. Perhaps a better technique would be to take a pre-rendered bubble and draw it to the screen. Get the `bubble.png` graphic from the code bundle and import it into `res/drawable/bubble.png`.

4. Open up the `BubblesView.java` source file. We will store a bitmap associated with the bubble resource. Add the following import lines at the top of the file:

```
import android.graphics.Bitmap;
import android.graphics.BitmapFactory;
```

These are all the extra classes we need to import here. We won't be using the bitmap apart from to pass it to the constructor of `Bubble`.

Add the following member variable to `BubblesView`:

```
   private Bitmap bubbleBitmap;
```

We'll assign this next.

5. In the constructor for `BubblesView`, add the following line:

```
      bubbleBitmap =
         BitmapFactory.decodeResource(
            context.getResources(),
            R.drawable.bubble);
```

This is a straightforward way of resource retrieval. Now to use it!

6. In the method `randomlyAddBubbles`, add the following argument to the constructor for bubbles:

```
   public void randomlyAddBubbles(
      int screenWidth,
      int screenHeight)
{
      if (Math.random()>BUBBLE_FREQUENCY) return;
      bubbles.add(
            new Bubble(
                  (int) (screenWidth*Math.random()),
                  screenHeight+Bubble.RADIUS,
                  (int) ((Bubble.MAX_SPEED-0.1)*Math.
random()+0.1),
                  bubbleBitmap));
}
```

This will show up as an error if you're running Eclipse, because we have not added this method to `Bubble` yet. Let's go there now.

7. Open up `Bubble.java` and add the import for `Bitmap` at the top as follows:

```
import android.graphics.Bitmap;
```

Also, add a member variable inside the class to hold the bubble graphic as follows:

```
private Bitmap bubbleBitmap;
```

Add it as an argument in the `Bubble` constructor, shown as follows:

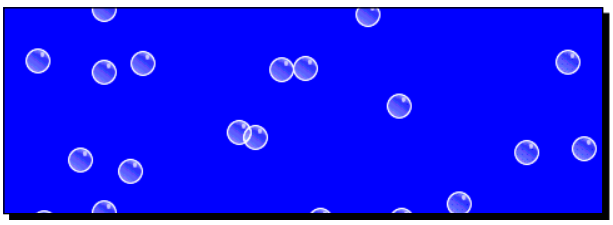
```
   public Bubble (
      float x,
      float y,
      float speed,
      Bitmap bubbleBitmap)
{
      this.x = x;
      this.y = y;
      this.speed = Math.max(speed, MIN_SPEED);
      this.bubbleBitmap = bubbleBitmap;
}
```

8. Now we have a picture of a bubble that we can use from the `Bubble draw` method. Change the `draw` method to look like the following:

```
   public void draw(Canvas c)
{
      c.drawBitmap(
         bubbleBitmap,
         x-RADIUS,
         y-RADIUS,
         bubblePaint);
}
```

Now we are animating bitmaps rather than those slightly weird lines of text!

9. Build the application and run it to see if it looks like the following:



See how the bitmap graphic allows us to easily create something with a bit more detail than just an ordinary `drawCircle` operation on a `Canvas`.

10. I've had another idea of how to make our scene look better. Bubbles form distorted shapes as they pass through the fluid and move in. How about making squishy bubbles that distort as they go along?

To achieve this, we will need to add a little bit more animation to our bubbles' `move` and `draw` methods. Open up `Bubble.java` again, and add the following member variable:

```
private float amountOfWobble = 0;
```

A bubble will update this value whenever a `move` is requested. The amount of wobble will be controlled by a constant value, so add the following constant into `Bubble.java`:

```
public static final float WOBBLE_RATE = 1/40;
```

You will see how this affects the wobble in just a minute. We also require control over the magnitude of the wobble, so add the following final constant too:

```
public static final int WOBBLE_AMOUNT = 3;
```

11. Now, let's update the `move` method. Recall that this method is called during the `calculateDisplay` portion of the game loop. I've highlighted the change to make in `move()` as follows:

```
public void move()
{
      y -= speed;
   amountOfWobble = (float)Math.sin (y*WOBBLE_RATE);
}
```

Now you can see the effect the wobble rate has, and also how the `amountOfWobble` relates to the bubble's position on the screen.

12. Next we want to draw an oval object in the Bubble's `draw` method. Replace the bitmap-drawing code with the following new oval-drawing routine:

```
   public void draw(Canvas c)
{
      c.drawOval(
         new RectF(
            x-RADIUS-WOBBLE_AMOUNT*amountOfWobble,
            y-RADIUS+WOBBLE_AMOUNT*amountOfWobble,
            x+RADIUS+WOBBLE_AMOUNT*amountOfWobble,
            y+RADIUS-WOBBLE_AMOUNT*amountOfWobble),
         bubblePaint);
}
```

The oval is drawn in a rectangular space, and we distort the rectangle a little bit each time because we are using the `amountOfWobble` variable that is set in `move`.

Notice that, although it would be just as easy to perform the wobbling calculation in the `draw` method, we do not do this. By keeping the `calculateDisplay` portion of the code separate from the `drawScreen` part of the code, we make it easier to debug and work on our code later.

13. When we were drawing circular bubbles earlier, they were pixelated and hollow. Let us try a different approach this time. We want them to be filled, smooth, and translucent.

Where do you think we want to make this change? Go on, have a guess.

That's right, we want to change the `bubblePaint` object, so that it incorporates these changes. Change the static block in `Bubble.java`, so that it looks like the code block given as follows:

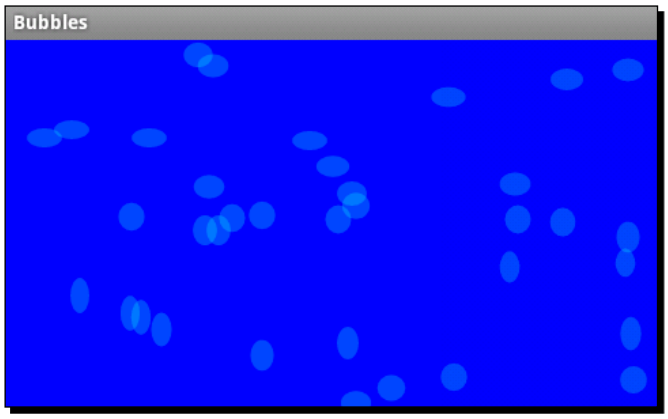
```
   static
{
      bubblePaint.setStyle(Paint.Style.FILL);
      bubblePaint.setColor(Color.CYAN);
      bubblePaint.setAlpha(66);
      bubblePaint.setAntiAlias(true);
}
```

Reading this line-by-line should be fairly intuitive. We want our bubbles to be as follows:

- filled
- cyan
- translucent (low alpha value)
- anti-aliased (smooth)

The `bubblePaint` is already applied to the oval that we are drawing, so we should see the effects next time we run the application.

14. Well, what are you waiting for? Build and run the application and see the following on your device:



What just happened?

In this example, we made use of a few more of the Canvas' features to create a programmatic animation.

We saw that the `Canvas` supports drawing bitmaps to the screen, and this allows us to combine programmatic animation with prepared images.

We also saw that the `Paint` class has a significant secondary effect on our animation graphics. With it, we can make our graphics translucent, smooth, filled, or hollow.

We were careful to make programmatic changes to our code during the `calculateDisplay` phase of our game loop. Make sure that the code you write goes somewhere you would expect to find it later, or else you will end up getting confused later!

For reference, here are a few of the handy ways that you can draw on a surface.

Getting to know the drawing tools in Canvas

These are methods that you can call on any `Canvas` to draw your animation programmatically. All drawing methods take `Paint` as an argument, so you can modify the way that the shape is drawn.

Canvases are not just useful for surfaces, you can use them to write to bitmaps, and they are used for updating view graphics too. For more information, check out the API documentation, which can be found at the following site:

`http://developer.android.com/reference/android/graphics/Canvas.html`

drawBitmap and drawPicture

These are handy ways to draw a picture, based on a `Picture` or `Bitmap` object. This is how we drew the bitmapped bubble display you saw previously.

drawCircle

This draws a circle given a centre coordinate and a radius value. We used this in the very first example in this chapter.

drawColor and drawPaint

These fill the whole `Canvas` with a single color. This value can have an alpha component, so if you want to put a `SurfaceView` over another view you can make translucent effects.

drawLine and drawLines

These help in drawing a straight line between two points, or a series of straight lines based on a list of start and end points.

drawOval and drawArc

`drawOval` draws an oval within a bounding rectangle shape.

`drawArc` draws a segment of an oval, determined by the shape of an oval and the start and stop angles of the arc.

drawPath

A `Path` is a special vector-like way of drawing lines and curves. If you need curvy lines, a `Path` is your friend. For more information, have a look at the following:

`http://developer.android.com/reference/android/graphics/Path.html`

drawRect and drawRoundRect

These help in drawing a rectangle to the screen, based on the x values of the right and left sides, and the y values of the top and bottom. `drawRoundRect` also lets you specify how much you would like the edges of the rectangle to be rounded off.

drawText and drawTextOnPath

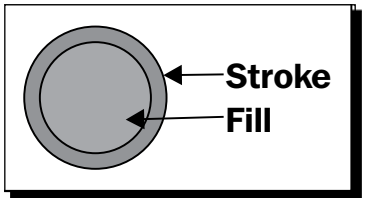
We saw how `drawText` could be used to draw text to the screen, in the previous example.

`drawTextOnPath` combines this technique with the ability to specify a vector-like line. Using this method, you can make your text move along a curve, and twist around. It can look very cool.

Using Paint effects

The `Paint` class is a little simpler than the `Canvas`, but it works just as hard. In fact, it offers a whole swathe of options for describing the way that something is painted on the screen. Here are a few methods that you will find useful when constructing a `Paint` object.

`Paint` objects have a notion of the stroke of the paint, which is what lines are, and the fill of the paint, which is what a filled shape such as a rectangle or circle is filled with.



You can choose whether you want to draw the outline of a shape, or fill the inside, using `setStyle` given as follows.

setAlpha

This sets the translucency value of the paint, from 0 (invisible) to 255 (solid).

setAntiAlias

Setting this to true removes any jagged edges that occur when painting a shape or line. It does not affect bitmap graphics.

setColor

This sets the color of the paint that we are working with, including its alpha.

setStrokeCap

This sets the shape of the "pen" that is used for drawing the stroke part of an object. It can be square or round.

setStrokeWidth

This is useful when drawing lines. It allows you to choose how thick you want them to be.

setStyle

This is where you can specify whether you want your shape to draw the outline `STROKE`, the middle `FILL`, or both `FILL_AND_STROKE`.

setTextAlign

This is used for text operations to choose whether to align the text to the left, center, or right.

setTextScaleX

This is used for text operations, how much to stretch the text along the x axis.

setTextSize

This is used for text operations to specify the size of the text.

setTypeface

This is used for text operations to choose the typeface for writing text.

Frame scheduling

In the previous examples we have assumed that we want a fixed frame rate, that is, if we finish the animation at a reasonable pace, we put the game loop to sleep and free up processor time for any other threads that might need it.

There are cases, particularly with game programming, when it is better to increase the frame rate whenever we can. A smoother screen is always better in terms of user experience. It is generally a good idea when we want a responsive animation, and when we know that most of our animation code is happening within one `Thread`.

You should use this technique sparingly in your applications, and consider who will be using your application. It not only makes other tasks run more slowly, it also drains battery life. Game players are used to games that drain battery power, but ordinary users will get upset if their battery life is drained for no good reason.

Time for action – creating smooth game loops

We are going to liberate our animation from the constraints of frame-by-frame updating. We will keep the notion that there are fixed, logical frames, but we will update the `calculateDisplay` portion of the code to update only a fraction of a frame at once.

Keeping the idea of fixed logical frames is not necessary, but we need some way of keeping the pace of the animation constant, and it shows the link between this animation and the previous one.

1. Firstly, we are going to update the game loop. Open up `BubblesView.java`.

   Navigate to the `run` method in the `GameLoop` class and replace it with the following:

```
public void run()
{
Canvas canvas = null;
long thisFrameTime;
long lastFrameTime = System.currentTimeMillis();
float framesSinceLastFrame = 0;
final SurfaceHolder surfaceHolder =
BubblesView.this.surfaceHolder;
while (running)
{
try
{
canvas = surfaceHolder.lockCanvas();
synchronized (surfaceHolder)
{
if (canvas != null) {
drawScreen(canvas);
calculateDisplay(canvas,framesSinceLastFrame);
}
}
}
finally
{
if (canvas != null)
surfaceHolder.unlockCanvasAndPost(canvas);
}
thisFrameTime = System.currentTimeMillis();
framesSinceLastFrame = (float)
(thisFrameTime - lastFrameTime)/msPerFrame;
lastFrameTime = thisFrameTime;
}
}
```

2. Next, as you might expect, we need to update the `calculateDisplay` method. The changes you need to make are very simple and are shown as follows:

```
   private void calculateDisplay(
      Canvas c,
      float numberOfFrames)
{
      randomlyAddBubbles(
```

```
            c.getWidth(),
            c.getHeight(),
            numberOfFrames);
        LinkedList<Bubble> bubblesToRemove = new
LinkedList<Bubble>();
        for (Bubble bubble : bubbles)
{
            bubble.move(numberOfFrames);
            if (bubble.outOfRange())
            bubblesToRemove.add(bubble);
}
        for (Bubble bubble : bubblesToRemove)
{
            bubbles.remove(bubble);
}
}
```

3. Navigate to the `randomlyAddBubbles` method and change the return line at the top of the method to the following:

```
   public void randomlyAddBubbles(
      int screenWidth,
      int screenHeight,
      float numFrames)
{
      if (Math.random()>BUBBLE_FREQUENCY*numFrames) return;
```

4. Next, open up `Bubble.java`. We will need to update its `move` method to handle the fractional frames too. Navigate to the `move` method and modify it as follows:

```
   public void move(float numFrames)
{
      y -= speed*numFrames;
      amountOfWobble = (float)Math.sin (y*WOBBLE_RATE);
}
```

5. The game loop has been updated, the `calculateDisplay` portion now includes a dynamic calculation. Build and run your application. Do you notice any additional smoothness?

What just happened?

In this tutorial, we saw that you do not need to stop and wait for a logical frame; you can keep animating your code and let the frame rate follow you. It makes the game loop as smooth as possible, but because our thread is running more frequently it can take up a lot more of your device's resources.

Adjusting the frame duration

In step 1, we introduced some new values. There is a `thisFrameTime` and a `lastFrameTime` value. These are measurements of the system clock taken between update cycles. The "frame" in this case refers to a fractional frame. `framesSinceLastFrame` will contain the fractional number of frames since the last frame.

By taking the difference between the `lastFrameTime` and `thisFrameTime` and dividing it by the number of milliseconds of a logical frame, we get the number of fractional frames that have elapsed. This is likely to be a small figure, such as 0.1.

The last thing we have done is passed the `framesSinceLastFrame` value as an argument of `calculateDisplay`. From now on, `calculateDisplay` needs to know how much it should update its data by.

In step 2, all we are doing is passing the number of frames down into the other methods in the `calculateDisplay` routine. Let's update them with sensible new behavior for fractional frames.

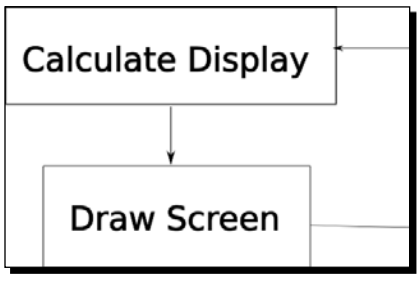
We had to adjust the method that randomly adds bubbles in step 3. Because the value `BUBBLE_FREQUENCY` is what determines the likelihood of a new bubble being created, it must now be adjusted to suit the fractional value of frames.

If, previously, we were calling this method 25 times per second, and now we are calling it 250 times per second, we need to adjust the `BUBBLE_FREQUENCY` by a factor of 10, or we will get ten times as many bubbles!

Similarly in step 4, you will notice that the speed is multiplied by the fraction of a frame that has passed. This makes sense if you think about it: if twice as many frames are being processed per second, then the `numFrames` value will be 0.5, and the bubble will move half as far as it used to. But it will now move twice in the same number of milliseconds that it used to move once.

Taking the wait out of the game loop

Our game loop flowchart now looks like the following:



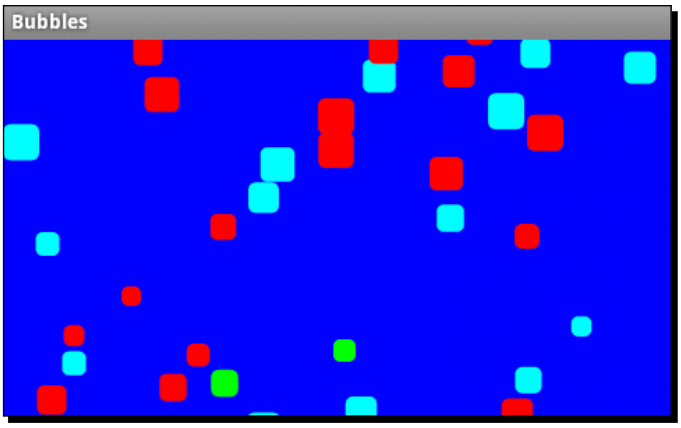
The observed increase in performance may vary depending on the application that you are writing, and on the devices that you are writing your software for. You may want to experiment with both types of game loop before you decide which is more effective for your application.

Pop quiz – surface animations

1. What would you call to get a `Canvas` object from a `SurfaceHolder`?
   a. `getCanvas()`
   b. `getSurface()`
   c. `lockCanvas()`
   d. `lockSurface()`
2. Why would you want to update a fixed number of frames per second?
   a. For simple animations, it uses less processor power
   b. It's faster
   c. It's smoother
3. What does the `setStyle` method on a `Paint` object change?
   a. The color and alpha of the paint
   b. The text formatting
   c. The text typeface
   d. The stroke and fill of the paint
4. Why is it important that we do not write to a surface once it has been destroyed?
   a. It will not affect anything
   b. It will draw onto another application
   c. It will force close your application
5. What must you do at the start of every frame in a surface-based animation?
   a. Empty it of all graphics
   b. Empty the graphics, which you are about to change
   c. Nothing, it will already be empty
6. In the last tutorial, which part of the game loop did we optimize?
   a. `drawScreen`
   b. `calculateDisplay`
   c. `waitTillNextFrame`

Have a go hero – rainbow bubbles!

The bubble-drawing routine is rather plain; who wants to look at ordinary bubbles? Change our animation, so that each bubble has a random or semi-random color.



If you are feeling particularly adventurous, give each bubble a different shape, too! In fact, try to make the animation look as crazy as you like.

Summary

In this chapter, we learned how to create animations programmatically, using surfaces and their built-in drawing tools. We used a game loop pattern to update the display.

Specifically, we covered the following:

- Game loops, and the components of a handmade application
- How a surface interfaces with the views system
- How we can get a surface using `SurfaceHandlers`
- Using `Canvas` and `Paint` methods to create a wealth of different graphical elements
- Optimizing the game loop to get more frames per second

We also learned that we could take advantage of a surface and a game loop to get better performance, or to make our animation more efficient.

Our skills with surfaces will come in handy in the next chapter, which is all about creating live wallpapers. These are pretty visualizations that sit behind the icons on your home screen and are ideal for programmatic animations.

8
Live Wallpapers

In this chapter, we will look at a particular type of Android application that is purposefully designed to contain an animation. A live wallpaper is a type of Android service that provides an animated graphic to sit behind your icons on your device's home screen. We will make use of the `SurfaceView` *animation that we learned about in the previous chapter.*

In this chapter, we shall:

- Create an application that provides a live wallpaper service
- Use a `SurfaceView` to implement the animation
- Add interactive features to our live wallpaper

Let us start by creating our first live wallpaper.

Creating a live wallpaper

The live wallpaper is a service as opposed to all of the other applications that we have created in this book, which are activities. This is because it does not run on its own, but as part of another application, usually the Android home screen.

If you skipped the previous chapter, don't worry. I will provide all the files you need to do this tutorial. If you did the *Have A Go Hero* section, great! You can use your custom bubbles animation in this tutorial.

Time for action – making our first live wallpaper

Remember the bubbles animation that we made in the previous chapter? We made a programmatic animation where we drew bubbles onto the screen. In this chapter, we will take that animation and make it into a live wallpaper for your device.

We will go through all the stages required to get a live wallpaper animation appearing on your device's Android home screen.

1. Create a new Android project with the following settings:
   - **Project name:** `BubblesWallpaper`
   - **Build Target:** `Android 3.0`
   - **Application name:** `BubblesWallpaper`
   - **Package name:** `com.packt.animation.bubbleswallpaper`
   - Do not specify a default activity

You may be wondering why we are not specifying the default activity. It's because a live wallpaper is not an activity on its own but a service that it provides to the Android home screen. That's right, this application does not need an activity to run. This also means that you won't see an application launch when you press Run in Eclipse.

2. Instead of an activity, we are going to provide a service. In the `AndroidManifest.xml` file in your project directory, add the following item inside the `<android:application>` tags:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
  xmlns:android="http://schemas.android.com/apk/res/android"
  package=»com.packt.animation.bubbleswallpaper»
  android:versionCode="1"
  android:versionName="1.0">
  <uses-sdk android:minSdkVersion="11" />
  <application
    android:icon="@drawable/icon"
    android:label="@string/app_name">
 <service
   android:name=»BubbleWallpaperService»
   android:enabled=»true»
   android:icon=»@drawable/icon»
   android:label=»@string/app_name»
   android:permission=
```

```
    "android.permission.BIND_WALLPAPER">
    <intent-filter android:priority="1">
    <action
      android:name="android.service.wallpaper.WallpaperService"
    />
  </intent-filter>
  <meta-data
    android:name="android.service.wallpaper"
    android:resource="@xml/wallpaper" />
  </service>
    </application>
</manifest>
```

The `meta-data` element refers to an XML `<wallpaper>` definition that doesn't exist yet.

3. So let us create it! Create a new file in `res/xml/` called `res/xml/wallpaper.xml`, and give it the following content:

```
<?xml version="1.0" encoding="utf-8"?>
<wallpaper
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:description="@string/wallpaperDescription"
  android:author="@string/author"
  android:thumbnail="@drawable/icon" />
```

This is what Android will use to describe the wallpaper to the user. In this instance, it is going to look like the following screenshot:

But you will probably have noticed that we haven't defined those strings yet; the description will not look like the screenshot until we have done the next step.

4. Let's go to the file `res/values/strings.xml` and define them now. Your `strings.xml` file should look like the following lines of code:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string name="app_name">Bubbles Wallpaper</string>
  <string name="author">Your Name</string>
  <string name="wallpaperDescription">Lovely Bubbles</string>
</resources>
```

These strings will be resolved when needed, using the `@string/name` reference types mentioned above in the code.

You may notice a compile error because `res/layout/main.xml` cannot find the string `@string/hello`. You can delete `main.xml` as it will not be required for this tutorial.

5. Now we need to actually define our service. You will recall from *step 2* that we named it `BubbleWallpaperService`, so create a new Java file called `BubbleWallpaperService.java`. In it, we will add the following import definitions:

```
import android.service.wallpaper.WallpaperService;
import android.view.SurfaceHolder;
```

The first of these is a prototype for making live wallpapers; you will see how we subclass it in a minute. The second is so that we can acquire a surface to draw our live wallpaper onto.

6. Now, let us create the class itself. Add the following lines:

```
public class BubbleWallpaperService extends WallpaperService {
  public Engine onCreateEngine() {
    return null;
  }
}
```

7. Before we implement the engine, we need to define the animation that it is going to use. Here we have two choices:

- If you completed *Chapter 7*, *2D Graphics with Surfaces*, you can copy the files listed next from the `Bubbles` project that you made.
- If you did not complete the chapter, you will need to get the files from the code bundle. I may have made the bubbles a bit trippy.

The files you will need are these:

- `Bubble.java`
- `BubblesView.java`

Copy them to the `src/com/packt/animation/bubbleswallpaper` directory. At the top of each file, change the package name to `com.packt.animation.bubbleswallpaper`.

If you are using the `BubblesView.java` from the previous chapter, you will need to change its constructor. Open up `BubblesView.java` and look for these lines:

```
public BubblesView(Context context, AttributeSet attrs) {
  super(context, attrs);
```

If these two lines exist, you should change them to read:

```
public BubblesView(Context context) {
  super(context);
```

We don't need to worry about Attributes in this example.

8. Now we need to add an engine that calls our bubble graphic. We will create it as a private class within our wallpaper service so that it is all kept together. In `BubbleWallpaperService.java`, add the highlighted new class:

```
package com.packt.animation.bubbleswallpaper;
import android.service.wallpaper.WallpaperService;
import android.view.SurfaceHolder;
public class BubbleWallpaperService extends WallpaperService {
  @Override
  public Engine onCreateEngine() {
return new BubbleWallpaperEngine();
  }
  private class BubbleWallpaperEngine extends Engine {
private BubblesView bubbles;
private SurfaceHolder surfaceHolder;
```

The `BubblesView` is our bubbles animation. All it needs is a valid `SurfaceHolder`, and it will draw its animation on it.

```
public void onCreate(SurfaceHolder surfaceHolder) {
  bubbles = new BubblesView(getApplicationContext());
  this.surfaceHolder = surfaceHolder;
  bubbles.surfaceCreated(surfaceHolder);
}
```

Note that, although we are using a `SurfaceView` in this instance, our animation does not need to be a view. It can be any object that writes to a surface.

```
public void onDestroy() {
  bubbles.surfaceDestroyed(surfaceHolder);
}
public void onSurfaceChanged(
  SurfaceHolder surfaceHolder,
  int format,
  int width,
  int height) {
this.surfaceHolder = surfaceHolder;
bubbles.surfaceChanged(
  surfaceHolder,
  format,
```

```
        width,
        height);
    }
    public void onVisibilityChanged(boolean visible) {
      if (visible) {
        bubbles.startAnimation();
      } else {
        bubbles.stopAnimation();
      }
    }
  }
}
```

9. There we are, all ready to run! This will be a bit more complex to launch than our usual applications, because we have not created an activity to run on the device. First you must deploy the application to your device as usual.

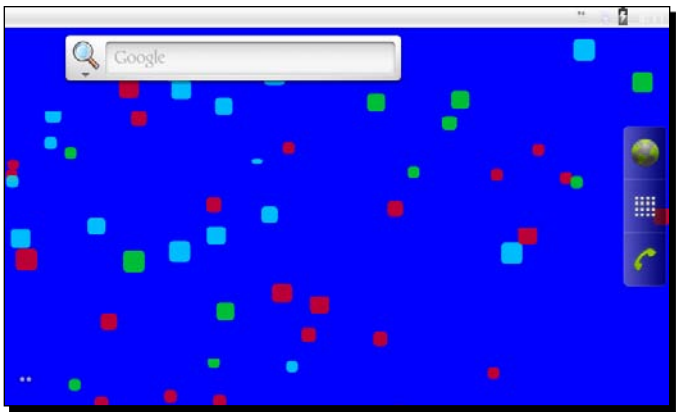
10. Next, navigate to the Android home screen on the device. Find an area that does not have any widgets or icons on it, and touch and hold the screen. You will see a new dialog called **Add to Home screen** appear.

11. Select **Wallpapers**.

12. Select **Live Wallpapers**.

13. Look for the entry marked **Bubbles Wallpaper**. Notice how it appears, based on the `wallpaper.xml` that we defined in *step 3*. Select the **Bubbles Wallpaper** entry.

14. You should see our wallpaper animation (the one in the code bundle is blue with multi-colored square bubbles), along with a button that says **Set wallpaper**. Select the **Set wallpaper** button.



You should then see your animated bubbles appear as a moving wallpaper behind the Android home screen!

What just happened?

Here we created a live wallpaper, which is an animation that runs behind the Android home screen to make your mobile desktop look pretty. We saw that there are several parts to this recipe. We need to:

- Declare our wallpaper in the `AndroidManifest.xml` by associating it with an Intent
- Describe our wallpaper by specifying an XML `<wallpaper>` resource that Android can retrieve
- Subclass the `WallpaperService` class to be able to respond correctly to the `android.service.WALLPAPER_SERVICE` Intent
- Subclass the `WallpaperService.Engine` class to associate the surface that describes the wallpaper on the screen with the animation that we want to appear on it
- Have some sort of surface-based animation to display

Declaring a live wallpaper

In *step 2*, we declared to the `Service` class that we are going to provide `BubbleWallpaperService`. We also request a security permission that is required for our wallpaper to work, and register the Intent type that our service will respond to. When the Android device sends out an `android.service.wallpaper.WallpaperService` message, this application will respond to it. This is how our device will know that the new application is a wallpaper.

How live wallpapers appear

As you saw in the tutorial, the live wallpaper does not run as a standalone application. Instead, we have to use another application, the Android home screen, as a host. This means that you should follow the following steps to launch your live wallpaper (repeated from the previous tutorial):

1. Navigate to the Android home screen on the device. Find an area that does not have any widgets or icons on it, and touch and hold the screen. You will see a new dialog called **Add to Home screen** appear.
2. Select **Wallpapers**.
3. Select **Live Wallpapers**.

4. Select the entry that relates to the live wallpaper that you are working on.
5. Select **Set wallpaper**.

Once you have created a live wallpaper and set it as your device's wallpaper, you do not usually have to repeat this process. If you download a newer version of the wallpaper, Android will automatically launch it once it has downloaded.

Developing live wallpapers in an activity

When you are developing a live wallpaper, you may find it difficult to debug. The live wallpaper does not launch in the same way that an activity launches; it is harder to connect to a debugger.

Because live wallpapers are based on a surface, you can easily test the animation part of your wallpaper, by creating a simple activity that creates a `SurfaceView`, and then passing the `SurfaceHolder` it receives to the `surfaceCreated` method in your development code.

If you need to revise your `SurfaceView` skills to do this, have a read through *Chapter 7*, *2D Graphics with Surfaces*, again.

The live wallpaper surface sits below all icons and buttons on the Android home screen, so it will not interfere with the user's ability to press buttons and select icons.

Understanding services

A service is something that belongs to one application, but that can be used in another. The service usually runs in a separate process, so that (for instance) even if your live wallpaper crashes, the Android home screen will continue to work.

Services can be created, destroyed, started, and stopped, by a calling application. You do not simply construct them as objects in the same way that you would create a local Java object.

You'll find more information about services in the Android reference documentation here:

`http://developer.android.com/reference/android/app/Service.html`

WallpaperService

Fortunately, when it comes to live wallpapers, the `WallpaperService` takes care of the hard work of providing a service for you. If your `WallpaperService` class subclasses `android.service.WallpaperService`, the only thing you need to do is provide a `WallpaperService.Engine`. That's it!

To do this, you must override the `onCreateEngine()` method in your `WallpaperService` subclass. Assuming that you have created a `WallpaperService.Engine` subclass, you can construct one here and return it as the result. Do not return null, as it will trigger a force close.

A bit like an activity, a service has entry points relating to its lifecycle, and the `WallpaperService` extends this in a way that is useful for live wallpapers. In the previous example, the `onCreate` method is called when the Android home screen has allocated a surface to draw our live wallpaper onto. So it is here that we create the `BubblesView` and pass it the surface that was created. See *step 8* for the exact code.

WallpaperService.Engine

This is where the real work gets done. Like services, a `WallpaperService.Engine` can be created, destroyed, updated, or paused. I have listed the common features that you will need to implement when you are creating a `WallpaperService.Engine`, as follows:

As with the `SurfaceView` in the previous chapter, if the surface is destroyed or changed, we want to make sure our animation object is notified. If it tries to write to a non-existent surface, our wallpaper will force close.

onCreate(SurfaceHolder surfaceHolder)

This will get called whenever the wallpaper is added to the Android home screen. It is a good place to construct the underlying animation object for this engine.

This is similar to the `surfaceCreated` method in the `SurfaceHolder.Callback` we implemented in *Chapter 7*, *2D Graphics with Surfaces*. In the previous example, we use the `onCreate` method as a pass-through to the `surfaceCreated` method in the underlying `SurfaceView`.

Note that the `SurfaceHolder` may not actually hold a surface at the time of construction. You should implement the following methods relating to surfaces, to ensure that the `SurfaceHolder` is correct.

onDestroy()

When this method gets called, your live wallpaper has been destroyed. Stop writing things to the surface, and exit any animation threads you may have running.

onSurfaceChanged (SurfaceHolder surfaceHolder, int format, int width, int height)

This method will be called whenever a structural change to the surface is made. You should make sure that your internal `SurfaceHolder` is updated to be consistent, and that any internal record that you keep of the dimensions and format of the surface are updated too.

This is similar to the `surfaceChanged` method in the `SurfaceHolder.Callback` that we implemented in the previous chapter. We pass through our data to this method in the underlying `SurfaceView` animation.

onVisibilityChanged (boolean visible)

The code in `onVisibilityChanged` simply pauses the animation when it is not visible. By not running the animation thread when we don't need to, we help to keep the user's device running efficiently. The Android API documentation makes it very clear that our live wallpaper should not consume any processor time while invisible, so make sure that your animations don't!

In the previous example, we used `onVisibilityChanged` to start and terminate the game loop in the underlying animation.

Pop quiz – live wallpapers

1. Which class do you subclass to get a live wallpaper service?
   a. `android.service.wallpaper.WallpaperService`
   b. `android.wallpaper.LiveWallpaper`
   c. `android.service.LiveWallpaperService`
2. What is the XML root element for defining a live wallpaper so that the Android home screen can see it?
   a. `<LiveWallpaper>`
   b. `<Wallpaper>`
   c. `<WallpaperInfo>`
3. What kind of animations can you show on a live wallpaper?
   a. View-based animations
   b. `SurfaceView`-based animations
   c. Any animation that can be applied to a surface

4. .Which class do you subclass to implement your animation?

   a. `WallpaperService.Animation`

   b. `WallpaperService.Engine`

   c. `WallpaperEngine`

Adding interactivity to live wallpaper

Something else that we can do with our animations is to make them interactive.
The `WallpaperService.Engine` can receive touch events from the Android home screen, and that allows us to make interactive animations.

Time for action – making soapy fingers

Wouldn't it be neat if, whenever we touched the Android home screen, a few more bubbles appeared around where our finger touched the screen? Well, that's what we're going to make next!

In the following example, we will see how to register our ability to handle touch events, how to implement the touch event handlers, and how they will appear on the screen.

1. First of all, we need to extend our `BubblesView` animation to support adding bubbles interactively. We will do this by implementing a method that can handle `MotionEvents` and create some extra bubbles.

   Open up `BubblesView.java` and add the following import:

   ```
   import android.view.MotionEvent;
   ```

   This will allow us to receive motion events from outside.

2. Next, let us create an `onTouchEvent` handler in `BubblesView`. Create it inside the class body and make it look like the following block of code:

   ```
   public boolean onTouchEvent(MotionEvent event) {
     boolean handled = false;
     if (event.getAction()==MotionEvent.ACTION_DOWN) {
       createSomeBubbles(event.getX(),event.getY());
       handled = true;
     }
     return handled;
   }
   ```

Next, we need to update our constants. Add the following constant values to `BubblesView`; the `BUBBLE_TOUCH_RADIUS` will be the area around the finger press where bubbles can form. The `BUBBLE_TOUCH_QUANTITY` is how many will form in one press.

```
private float BUBBLE_TOUCH_RADIUS = 30;
private int BUBBLE_TOUCH_QUANTITY = 7;
```

While we are working with the constant values, we should change the frequency with which bubbles come from the bottom of the screen. Edit the `BUBBLE_FREQUENCY` value, so that it appears as follows:

```
private float BUBBLE_FREQUENCY = 0.03f;
```

This will ensure that the new bubbles stand out and are easy to notice.

3. Now we are ready to implement our method to create bubbles. In `BubblesView` we will create a new method to randomly add a few bubbles to a given coordinate.

In the `BubblesView` class, add the following method. Some of you who are using the bitmap bubbles from the previous chapter might get an error, but don't worry!

```
private void createSomeBubbles(float x, float y) {
  for (
    int numBubbles = 0;
    numBubbles < BUBBLE_TOUCH_QUANTITY;
    ++numBubbles) {
    synchronized(bubbles) {
      bubbles.add(
        new Bubble(
          (int)
            (2*BUBBLE_TOUCH_RADIUS*Math.random() -
            BUBBLE_TOUCH_RADIUS + x),
          (int)
            (2*BUBBLE_TOUCH_RADIUS*Math.random() -
            BUBBLE_TOUCH_RADIUS + y),
          (int)
            ((Bubble.MAX_SPEED-0.1)*Math.random()+0.1)
        )
      );
    }
  }
```

For those of you who have an extra parameter in their `Bubble` constructor for a bitmap, change the new `Bubble` constructor to read:

```
new Bubble(
          (int)
            (2*BUBBLE_TOUCH_RADIUS*Math.random() -
```

```
        BUBBLE_TOUCH_RADIUS + x),
      (int)
        (2*BUBBLE_TOUCH_RADIUS*Math.random() -
        BUBBLE_TOUCH_RADIUS + y),
      (int)
        ((Bubble.MAX_SPEED-0.1)*Math.random()+0.1),
        bubbleBitmap
    )
```

This creates a set of `BUBBLE_TOUCH_QUANTITY` bubbles, with coordinates somewhere random near the given `x` and `y` coordinates.

Notice that we have included a synchronized statement around the `bubbles.add` method. This is because we are no longer working purely in a single-thread game loop—new bubbles can be created from outside the game loop thread and this must not cause any new crashes. By calling `synchronized`, we ensure that `bubbles` is only accessed by one thread at a time.

We must always synchronize `LinkedLists` when they are used in a multi-threaded fashion, as they can throw a `ConcurrentModificationException` if they are changed while they are being iterated over.

4. Add synchronized statements in all of the methods shown as follows:

In `randomlyAddBubbles`:

```
public void randomlyAddBubbles(
  int screenWidth,
  int screenHeight) {
    if (Math.random()>BUBBLE_FREQUENCY) return;
    synchronized (bubbles) {
      bubbles.add(
        new Bubble(
          (int) (screenWidth*Math.random()),
          screenHeight+Bubble.RADIUS,
          (int) ((Bubble.MAX_SPEED-0.1)*Math.random()+0.1)
        )
      );
    }
  }
```

In `drawScreen`:

```
private void drawScreen(Canvas c) {
  c.drawPaint(backgroundPaint);
  synchronized (bubbles) {
    for (Bubble bubble : bubbles) {
```

```
        bubble.draw(c);
      }
    }
  }
```

In `calculateDisplay`:

```
  private void calculateDisplay(Canvas c) {
    randomlyAddBubbles(c.getWidth(),c.getHeight());
    LinkedList<Bubble> bubblesToRemove = new LinkedList<Bubble>();
synchronized (bubbles) {
      for (Bubble bubble : bubbles) {
        bubble.move();
        if (bubble.outOfRange())
          bubblesToRemove.add(bubble);
      }
      for (Bubble bubble : bubblesToRemove) {
        bubbles.remove(bubble);
      }
}
  }
```

Note that, if you are using the version of the `Bubble` animation from *Chapter 7*, *2D Graphics with Surfaces*, that has dynamic frame timing (from the last *Time for action - creating smooth game loops* section), you should replace the line `bubble.move()` with:

```
        bubble.move(numberOfFrames);
```

5. Now we have an interactive animation. We want to include the interactive portion in our `BubbleWallpaperEngine`, so that it responds to `MotionEvents` received by the Android home screen.

Open up `BubbleWallpaperService.java` and add the following import:

```
import android.view.MotionEvent;
```

It is much the same as we did in `BubblesView.java`, but here we only need to know it so that it can be passed into `BubblesView`.

6. We also need to make sure that our `BubbleWallpaperEngine` is registered to receive touch events; by default, live wallpapers do not receive them.

Navigate to inside the `BubbleWallpaperEngine` class. Inside the `onCreate` method, add the following lines:

```
      public void onCreate(SurfaceHolder surfaceHolder) {
        bubbles = new BubblesView(getApplicationContext());
        this.surfaceHolder = surfaceHolder;
```

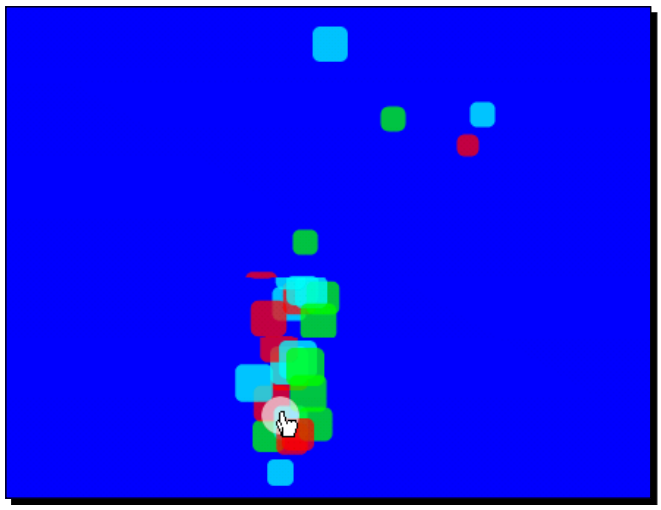
```
        bubbles.surfaceCreated(surfaceHolder);
    setTouchEventsEnabled(true);
    }
```

This is to make sure that we receive touch events through the `WallpaperService` interface.

7. Add the following new method to ensure that any events get passed through to the underlying `BubblesView` animation:

```
public void onTouchEvent(MotionEvent event) {
  bubbles.onTouchEvent(event);
}
```

8. Our live wallpaper is ready to test. Build and deploy the wallpaper to your device. If your wallpaper is not shown on the Android home screen, refer to the *How live wallpapers appear* section earlier, and follow the instructions to add it to the home screen.



Touch the screen and see how the interaction is passed through to your animated wallpaper.

What just happened?

We just made our animation interactive. In fact, there were several stages to making the animation respond to touch.

- First, we had to update our actual animation, so that it contained an interactive feature
- Then we had to ensure that our live wallpaper was registered to receive touch events from the Android home screen
- Finally, we had to take the messages received by the `BubbleWallpaperEngine` and use them to trigger the interactive feature of the wallpaper

Enabling WallpaperService.Engine interaction

There are two features in `WallpaperService.Engine` that you have to make use of in order to enable interaction.

1. You must call `setTouchEventsEnabled(true)` to start receiving touch events.
2. Override the method `onTouchEvent(MotionEvent event)` to receive motion events.

If both of these things are done, the `onTouchEvent` method will receive a `MotionEvent` every time the home screen is interacted with.

In the previous example, we set up an `onTouchEvent` method, which ensures that if our animation receives an event corresponding to a finger being pressed on the screen, it creates some bubbles with the `createSomeBubbles` method. The arguments to `createSomeBubbles` are the coordinates where the finger was pressed down. This is used to create a swarm of bubbles around the touched area.

Registering live wallpaper interaction

There are several different things that the `onTouchEvent` can receive in its `MotionEvent`, including the following:

- Pressing down and lifting up a touch
- Stroking the screen
- Multi-touch touches and strokes

In the example, we saw how to retrieve a single pressing down event, by testing for `MotionEvent.ACTION_DOWN`. Testing for other sorts of events is a similar process.

For more information about what you can do with a `MotionEvent`, visit the Android API documentation here:

`http://developer.android.com/reference/android/view/MotionEvent.html`

But remember that this is a generic class that handles many sorts of motion events. Only those events relating to touching the Android home screen are going to be received by your live wallpaper.

Pop quiz – interactivity

1. What kind of event does your live wallpaper receive when a user touches the screen?
   a. `ScreenTouchEvent`
   b. `MotionEvent`
   c. `AudioEvent`
2. What do you call when you want your wallpaper service to be registered to receive interaction messages?
   a. `setTouchEventsEnabled`
   b. `getTouchEvent`
   c. `enableTouchEvents`
3. Which of these returns `true` when the event is a finger being pressed on the screen?
   a. `event.getAction() == MotionEvent.ACTION_UP`
   b. `event.getAction() == MotionEvent.ACTION_MOVE`
   c. `event.getAction() == MotionEvent.ACTION_DOWN`

Have a go hero – popping bubbles

Everybody loves to pop bubbles. We've seen how to use live wallpaper interactivity to make bubbles appear on screen, so how can we use this approach to provide a different interaction? Let us change the bubbles wallpaper, so that you can pop bubbles by touching them.

To get you started, I'll show you the two methods that you will need to add in to the bubble wallpaper. The first one belongs in `Bubble.java`; it takes two co-ordinates as an argument and tells you whether the co-ordinates are inside the bubble. We will use this to determine whether a bubble is hit by a finger press.

```
public boolean isHit(float x, float y) {
  return
  x > this.x - RADIUS
  && y > this.y - RADIUS
  && x < this.x + RADIUS
  && y < this.y + RADIUS;
}
```

When a finger press is detected, we will need to loop over all of the bubbles on screen and detect whether they have been pressed. Add the following method to `BubblesView.java` in order to do this:

```
private void popBubbles(float x, float y) {
  LinkedList<Bubble> bubblesToRemove = new LinkedList<Bubble>();
  synchronized (bubbles) {
    for (Bubble bubble : bubbles) {
      if (bubble.isHit(x,y)) {
        bubblesToRemove.add(bubble);
      }
    }
    for (Bubble bubble : bubblesToRemove) {
      bubbles.remove(bubble);
    }
  }
}
```

Now all you will have to do is replace the call to `createSomeBubbles` with a call to `popBubbles`, and you're done!

Using live wallpaper preferences

Live wallpapers aren't necessarily completely self-contained. For instance, we have seen in the past two chapters that there are lots of different things in a surface-based animation that can be parameterized; we could make these parameters available to a user in the form of a preferences panel.

In fact, Android expects that you will want to do this, and provides an optional button on the live wallpaper browser to allow you to customize the wallpaper as you wish.

In the next example, we will see how to add preferences to a live wallpaper animation.

Time for action – configuring a live wallpaper

A live wallpaper can be configured by using an activity that stores preferences. When the preferences are changed, the live wallpaper is updated to reflect the new preferences.

The following are preferences that we can make available to the user to change:

- Choosing to have interactivity or not
- The number of bubbles that are likely to appear

So let us begin.

1. We are going to create a particular type of activity called a `PreferenceActivity`. To create a `PreferenceActivity`, we must first create an XML description of it. Create a new file in `res/xml` called `res/xml/wallpaperpreferences.xml`. In it, put the following block of code:

```
<?xml version="1.0" encoding="utf-8"?>
<PreferenceScreen
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:title="Bubble Wallpaper Preferences">
  <PreferenceCategory
    android:title="Interaction"
    android:order="1">
    <CheckBoxPreference
      android:title="Enable Interaction"
      android:key="isInteractive"
      android:defaultValue="true"
    />
  </PreferenceCategory>
  <PreferenceCategory
    android:title="Bubble Settings"
    android:order="2">
    <EditTextPreference
      android:title="Number of Bubbles"
      android:defaultValue="0.03"
      android:numeric="decimal"
      android:key="bubbleFrequency"
    />
  </PreferenceCategory>
</PreferenceScreen>
```

2. Next, we need to create a `PreferenceActivity` to use these preferences. Create a new Java file called `BubblesPreferences.java` with the following imports:

```
package com.packt.animation.bubbleswallpaper;
import android.os.Bundle;
import android.content.SharedPreferences;
import android.preference.CheckBoxPreference;
import android.preference.EditTextPreference;
import android.preference.PreferenceActivity;
```

3. Next, let us create our actual class:

```
public class BubblesPreferences extends PreferenceActivity {
}
```

   This, on its own, is obviously not enough!

4. To populate the activity with preferences, we should override the `onCreate` method, as we do with most activities. Inside our class, add the following `onCreate` method:

```
public class BubblesPreferences extends PreferenceActivity {
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    addPreferencesFromResource(R.xml.wallpaperpreferences);
  }
}
```

   Here we are only concerned with the `addPreferencesFromResource` method, which simply tells the `PreferenceActivity` where to look to find the preferences to display.

5. Now that we have an activity, we should add it to our application's manifest. Open up `AndroidManifest.xml` in the root directory of your project, and add the following XML within the `<application>` tags, at the same level as the existing `<service>` declaration.

```
<activity
  android:label="@string/app_prefs"
  android:name=".BubblesPreferences"
  android:exported="true">
</activity>
```

   This means that the Android home screen will be able to access the preferences activity, should it need to.

6. Open up the file `res/values/strings.xml`. We've just used a string called app_prefs, so let's add one amongst the other string definitions:

```
<string name="app_prefs">Bubbles Wallpaper Settings</string>
```

7. Now that we have made the application accessible, we need to tell the Android home screen what it is called. Open up `res/xml/wallpaper.xml` again, we will add a new line describing our preferences activity.

```
<?xml version="1.0" encoding="utf-8"?>
<wallpaper
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:description="@string/wallpaperDescription"
  android:author="@string/author"
  android:thumbnail="@drawable/icon"
   android:settingsActivity=
"com.packt.animation.bubbleswallpaper.BubblesPreferences" />
```

8. Let us update the `BubbleWallpaperEngine` to cope with this new behavior. Open up `BubblesWallpaperService.java`, and add the following new member variable to the `BubbleWallpaperEngine` class:

```
    private boolean isInteractive = true;
```

We will use this Boolean to specify whether to handle touch events in our wallpaper.

9. Again in the `BubbleWallpaperEngine` class, navigate to the `onTouchEvent` method and add a Boolean test as follows :

```
  public void onTouchEvent(MotionEvent event) {
    if (isInteractive) {
      bubbles.onTouchEvent(event);
    }
  }
```

Now we have a way to control whether or not touch events do anything to our live wallpaper.

You cannot simply toggle the `setTouchEventsEnabled` method, as there is no guarantee that the change will propagate to your `Service` communications with the Android home screen. It is better to ensure that touch events are always enabled at the `Service` level, and manually disable them using a Boolean test like the one above.

10. We want to be able to change the frequency with which bubbles appear. Open up the `BubblesView.java` class file and add the following accessor to `BubblesView`:

```
public void setFrequency(float bubbleFrequency) {
  BUBBLE_FREQUENCY = bubbleFrequency;
}
```

11. Next, we should add a new `readPreferences` method within the `BubbleWallpaperEngine` class to get the stored `Sharedpreferences`. Open up `BubblesWallpaperService.java` again and add the following imports at the top of the file:

```
import android.content.SharedPreferences;
import android.preference.PreferenceManager;
```

12. Now we can add the preference-reading method to the `BubbleWallpaperEngine` class:

```
private void readPreferences(SharedPreferences preferences) {
  if (preferences.contains("isInteractive")
    && preferences.contains("bubbleFrequency")) {
    isInteractive =
      preferences.getBoolean("isInteractive", true);
    float bubbleFrequency =
      Float.parseFloat(
      preferences.getString(
        "bubbleFrequency",
        "0.03"
      )
    );
    bubbles.setFrequency(bubbleFrequency);
  }
}
```

Notice that we have to parse the floating-point value of `bubbleFrequency`. Even though it is a floating-point value, because the interface of the `PreferenceActivity` is a string edit box, our preference will be a string too.

This new method updates the new `isInteractive` flag and the `setFrequency` method in our `BubblesView`.

13. Now to ensure that the `readPreferences` method gets called when our live wallpaper loads.

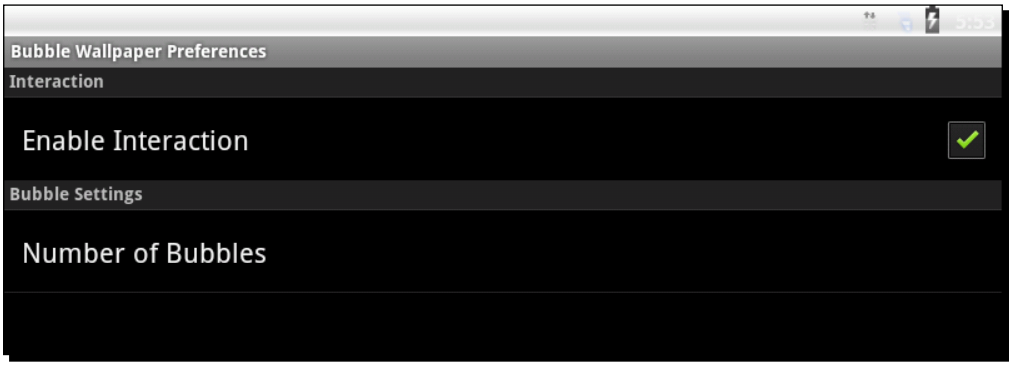
In the `onCreate` method in `BubbleWallpaperEngine`, add the following line to ensure that it gets run:

```
public void onCreate(SurfaceHolder surfaceHolder) {
  bubbles = new BubblesView(getApplicationContext());
```

```
        this.surfaceHolder = surfaceHolder;
        setTouchEventsEnabled(true);
        bubbles.surfaceCreated(surfaceHolder);
    SharedPreferences preferences =
      PreferenceManager.getDefaultSharedPreferences(
        getApplicationContext());
    readPreferences(preferences);
      }
```

Here we simply fetch the default preferences file and read it. This is the same `SharedPreferences` data that our `PreferenceActivity` will write.

14. Build and deploy your application.
15. Navigate to the Android home screen on the device. Find an area that does not have any widgets or icons on it, and touch and hold the screen. This is going to be similar to the way that you added the live wallpaper.
16. You will see a new dialog called **Add to Home screen** appear.
17. Select **Wallpapers**.
18. Select **Live Wallpapers**.
19. Select the entry that relates to the live wallpaper you are working on.
20. Select **Settings...** to see your configuration options.



Modify the settings and then press the **Back** button on your device. You will notice that the settings have not taken effect yet.

This is because the live wallpaper only reads the preferences in its `onCreate` method. If you press the **Back** button again and re-select the **Bubbles Wallpaper** to force a reload, you will see your settings have been applied.

What just happened?

We have created a preferences container for our live wallpaper. You will usually want to do this to provide options that your users might want to specify on the wallpaper that you are making. In this case, we want to allow the user to choose a few configuration options to make their wallpaper more suited to their taste.

Our preferences XML created two groups of information, each of which contains a single preference. One of the preferences is a checkbox called `isInteractive`, which we use to determine whether touch events should be used to create new bubbles. The other preference is a decimal value called `bubbleFrequency`, which we use to determine how many bubbles, on average, to create.

In Java, we used a few `Preferences` classes:

- `SharedPreferences` is a persistent storage method for saving and loading the preferences from disk.
- `PreferenceActivity` is a pre-made `Activity` class for displaying a list of editable preferences from XML.
- `EditTextPreference` and `CheckBoxPreference` are data classes that represent the preferences as they appear in the activity. We used them to retrieve the actual preference values that are set by the user.

The `PreferenceActivity` is capable of updating and storing the preferences from the preferences resource that we gave it, in this case, `R.xml.wallpaperpreferences`. It automatically stores and retrieves these values from the default `SharedPreferences` object as required.

In the XML description for the wallpaper, that is `res/xml/wallpaper.xml`, we associated our `PreferenceActivity` with the attribute `android:settingsActivity`. The live wallpaper browser in the Android home screen then used this to find the preferences. Note that we use the fully qualified Java name for the activity. It will make sure that the Android home screen does not search in the wrong place to find the preferences.

In the previous example, we treated `BUBBLE_FREQUENCY` as a constant, but in this example, we modify it by setting it from the preferences. Whenever `randomlyAddBubbles` is called, it will use this new frequency value.

It's currently not very tidy, because every time the preferences are updated for the wallpaper, the wallpaper needs to be reloaded. But there is a solution!

Updating preferences as soon as they are set

Android can notify us whenever our `SharedPreferences` objects are updated. This means that, as soon as the user customizes their preferences, our live wallpaper can be updated too. This is how most professionally made live wallpapers will behave. Let's find out how it's done!

Time for action – updating live wallpaper configuration

We want to add an event listener to the live wallpaper so that it is notified as soon as its configuration is changed. When it is changed, we are going to read the `SharedPreferences` object that has been updated.

1. Open up `BubblesWallpaperService.java` and navigate to the `BubbleWallpaperEngine` class. We will add an interface from `SharedPreferences`:

```
  private class BubbleWallpaperEngine extends Engine
implements SharedPreferences.OnSharedPreferenceChangeListener
{
```

   This allows us to listen for changes in the shared preferences, but first we must provide an implementation of the interface.

2. Add the following method to `BubbleWallpaperEngine` to complete the implementation:

```
public void onSharedPreferenceChanged(
  SharedPreferences sharedPreferences,
  String key) {
    readPreferences(sharedPreferences);
  }
```

3. So let us register our `BubbleWallpaperEngine` right now, as a listener to the `SharedPreferences` containing our `isInteractive` and `bubbleFrequency` values.

   Add the following member variable in `BubbleWallpaperEngine`:

```
private SharedPreferences preferences;
```

4. Navigate to the `onCreate` method in our `BubbleWallpaperEngine`. Make the highlighted changes:

```
  public void onCreate(SurfaceHolder surfaceHolder) {
    bubbles = new BubblesView(getApplicationContext());
    this.surfaceHolder = surfaceHolder;
```

```
        setTouchEventsEnabled(true);
        bubbles.surfaceCreated(surfaceHolder);
    preferences =
      PreferenceManager.getDefaultSharedPreferences(
        getApplicationContext());
        readPreferences(preferences);
    preferences.registerOnSharedPreferenceChangeListener(this);
      }
```

5. To clear the connection to the preferences object when it is no longer required, let us use the `onDestroy` method of `BubbleWallpaperEngine` to unregister the `OnSharedPreferenceChangeListener`. Add the following line:

```
    public void onDestroy() {
      bubbles.surfaceDestroyed(surfaceHolder);
  preferences.unregisterOnSharedPreferenceChangeListener(this);
    }
```

   This ensures that no notifications get sent to our live wallpaper, if it has been destroyed.

6. Now, build and deploy your application.

7. Navigate to the Android home screen on the device. Find an area that does not have any widgets or icons on it, and touch and hold the screen. You will see a new dialog called **Add to Home screen** appear.

8. Select **Wallpapers**.

9. Select **Live Wallpapers**.

10. Select the entry that relates to the live wallpaper that you are working on.

11. Select **Settings...** to see your configuration options.

12. Modify some settings, and press the **Back** button. Notice that the modifications that you make are immediately visible on the live wallpaper.

What just happened?

Now our live wallpaper will update itself automatically whenever the `BubblesPreferences` activity changes the preferences for our application. This is a much more intuitive way to interact with a live wallpaper, and it means that your users are not going to get confused by changes that do not immediately appear.

We wanted our live wallpaper to know when its preferences had been updated, and fortunately Android provides a notification system to allow us to be notified once a change has been added to the `SharedPreferences` object that we are interested in.

We registered a listener in the live wallpaper that gets called when the preferences are changed, and this meant that we could change the settings of our animation while it is running. By registering an `OnSharedPreferenceChangedListener`, our wallpaper activity's `onSharedPreferencesChanged` method will be called whenever a change is made to any `SharedPreferences` data that belongs to the `BubbleWallpaperEngine`.

Connecting our wallpaper to our prefereces

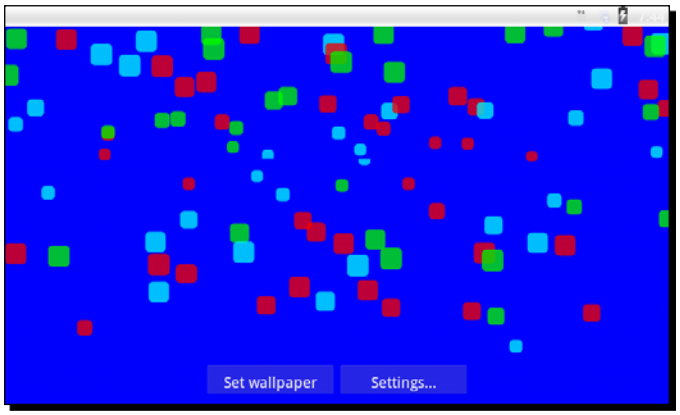
In the `onCreate` method for the `BubblesWallpaper`, we store the preferences object in our new member variable. We also registered our `BubbleWallpaperEngine` as an `OnSharedPreferenceChangedListener`. Now, whenever our `BubblesPreferences` makes a change to our `SharedPreferences`, the `BubbleWallpaperEngine` will update its values accordingly.

Disconnecting our preferences when our wallpaper exits

We manage the interaction between the `SharedPreferences` and the live wallpaper throughout the live wallpaper's lifecycle. By this I mean that we want to stop receiving change notifications as soon as our live wallpaper has been destroyed. So whenever our wallpaper's `onDestroy` method is called, we unregister its link to the `SharedPreferences`, by calling `unregisterOnSharedPreferenceChangeListener`.

How the user will see preferences

Preferences are usually specified in the same way that we choose the actual wallpaper. When browsing the live wallpaper, instead of immediately selecting **Set wallpaper**, you can choose **Settings...**, as shown in the following screenshot:



Once the **Settings...** button has been selected, you can make changes to your live wallpaper preferences, and see them applied immediately after you return to the wallpaper preview.

Storing preferences with SharedPreferences

The `SharedPreferences` object is a way of storing preferences between invocations of an application and between different activity and service instances running within it. It takes the form of a key-value store that can store `String`, `float`, `int`, and `Boolean` data. That is, you give it a `String` identifier, and it gives you a piece of data associated with the key, if it exists.

In the previous example, the string `isInteractive` is a key, and its value could be either `true` or `false`.

Reading from SharedPreferences

To read a key from a `SharedPreferences` object, you call:

- `getString(key)`
- `getInt(key)`
- `getLong(key)`
- `getFloat(key)`
- `getBoolean(key)`

You call these methods on the `SharedPreferences` instance that you are working with. Note that the type of key must be correct, or else you will get a `ClassCastException` at runtime.

Beware that when you are using `EditText` boxes in a `PreferenceActivity`, they will always set values as `String`, even if the values are strictly limited to integer or floating point type by setting `android:numeric`.

To test that a given key exists, you can call `contains(keyname)` which returns the Boolean answer.

Writing to SharedPreferences

`SharedPreferences` are usually read-only. In order to write to them, you must call `SharedPreferences.edit()` on your `SharedPreferences` instance, to retrieve a `SharedPreferences.Editor`.

Once you have done this, writing to `SharedPreferences` is taken care of by calling:

- `putString(key)`
- `putInt(key)`
- `putLong(key)`
- `putFloat(key)`
- `putBoolean(key)`

These methods will not update the `SharedPreferences` object immediately. Rather, you will need to call the `commit` method on your `SharedPreferences.Editor`. This will update all of the values in the underlying `SharedPreferences` object, and also call any `OnSharedPreferenceChangedListener` objects that are watching the `SharedPreferences`.

OnSharedPreferenceChangedListener

As we saw in the previous example, `OnSharedPreferenceChangedListener` objects can be associated with a given set of `SharedPreferences` and notified when they are changed.

The method that any `OnSharedPreferenceChangedListener` will need to implement is called `onSharedPreferenceChanged(sharedPreferences, key, value)`. The `sharedPreferences` object is the object that we are watching; the `key` and `value` are the preference key and value that have changed.

Composing preference XML

You define a set of preferences for use within a `PreferenceActivity` by writing them in XML. In the previous example, we declared two parameters that were later passed to the application's default `SharedPreferences` object.

You saw an example of the preference XML as `wallpaperpreferences.xml` in the previous example. Next is a list of the elements and attributes that we used to make the preferences screen in the previous example.

Defining preferences in XML

<PreferenceScreen>

`<PreferenceScreen>` is used as the root element to declare the overall screen, full of preferences. This usually contains a few individual preference values.

<PreferenceCategory>

`<PreferenceCategory>` is used to contain a group of preferences that are all associated somehow.

You can associate an `android:title` with this tag to explain the categorization.

<CheckBoxPreference>

`<CheckBoxPreference>` is a preference that has a `true` or `false` value, depending whether it is checked or unchecked respectively.

<EditTextPreference>

`<EditTextPreference>` is the basis for all preferences that take a keyboard input, including numeric values.

You can constrain the contents to a numeric value, by specifying the tag `android:numeric="integer"` or `android:numeric="decimal"`, but the underlying data will still be stored as a string.

Setting attributes on XML preferences

android:key

`android:key` is the key name of the preference. This is the name that will get added to the `SharedPreferences` object.

android:defaultValue

If our `PreferenceActivity` cannot find a previously defined value for a given key, it will show the `android:defaultValue` by default.

android:order

`android:order` is the order in which the given preference should appear on the screen. Lower values appear earlier.

Pop quiz – preferences for live wallpapers

1. If you don't want to restart your live wallpaper every time, you modify its preferences, what is the best thing to do?
   a. Nothing
   b. Keep refreshing the preferences file
   c. Define an `OnPreferencesChangedListener`
2. What is the wallpaper attribute for specifying an activity that provides preferences for your wallpaper?
   a. `settingsActivity`
   b. `wallpaperPreferences`
   c. `preferencesActivity`
3. Which class can you subclass to create a preferences activity?
   a. `android.settings.SettingsActivity`
   b. `android.preference.PreferenceActivity`
   c. `android.app.Customizer`

Have a go hero – doing more with preferences

We have seen some ways in which you can customize a live wallpaper, but how else might it be possible to configure the live wallpaper?

Try providing an extra option in the preferences, for setting the number of bubbles that get created when you touch the screen in interactive mode.

If you're feeling really adventurous, try to make all of the numeric members of `BubblesView parameterizable`. Do they all make sense to a user?

Summary

We learned a lot in this chapter about live wallpapers.

Specifically, we covered:

- Creating a service to provide a live wallpaper
- Implementing a live wallpaper as an animation on a surface
- Adding interaction to the live wallpaper
- Adding a set of configuration options, and linking it to the `Settings...` button in the Android wallpaper browser.

We ensured that the live wallpaper was responsible with respect to registering event listeners.

Now that we've learned about live wallpapers, we will move on to the final chapter, which is about making animations that are helpful to the user and work well on mobile devices.

9
Practicing Good Practice and Style

The contents of this chapter can be summed up in one simple rule: be helpful to the user.

On a mobile device, the user also expects us to be good citizens about the limited resources on the device. Don't try to show animations that are too big, don't distract from the important things, and above all don't hog the processor. This will not only make the rest of the phone work badly, it will drain the battery life too.

In this chapter, we will take a look at some of the ways in which you can make your animations behave in a helpful and responsible way. They are as follows:

- Use animation to explain behavior to a user
- Tidy up confusing animations
- Optimize an application for screen usage
- Monitor power usage
- Optimize an animation for power consumption

So let's get on with it.

Using focus and metaphor

Humans are a tool-using species, and we construct a mental model of how things are supposed to work. For instance, your mental model of a car might include, "pushing one pedal makes it go faster, and pushing the other makes it go slower". Good user interfaces provide simple user models that are easy to learn, and animation can help or hinder in that respect.

Focus is the thing that draws the user's attention toward something. Animation is an excellent device for directing a user's focus one way or another; when something is moving in our vicinity, we have the instinct to find out what it is. Have you ever been distracted by a moth or fly while you are working? It is hard to keep your eyes on your work if there is something dancing around your field of vision.

On your desktop computer, for instance, your word processor contains one of the simplest and most ancient computer animations: the blinking cursor. It is easy to see why; if you are writing a letter to your mother and you go away to make a cup of tea, how will you know where the cursor is when you come back? The cursor is just one tiny line in a sea of text, and a little animation brings it to the fore in a discreet way that compliments the visual style of the document.

Animation is, in fact, a more powerful tool than any other in creating focus. If you're as old as I am, you might recall that web pages used to use `<marquee>` and `<blink>` tags a lot. Eventually this died out, because it was generally regarded as horrible and distracting – sure you want your users to look at the important text, and sure animation is effective, but then you would be put off while reading the rest of the webpage. And it was ugly. Did people complain in the same way about bold text, or different colors? Of course not! Your attention is not caught in such a powerful way. Be careful!

While focus directs your attention, metaphor explains what you are looking at. It is not the only way to explain things, and one of the worst things you can do is to try to shoehorn a meaningless metaphor into a user interface. However, a consistent metaphor helps users navigate the features of a document.

Using the example of the word processor again, think about the menus. Menus in restaurants were, of course, the inspiration: you sit down in the restaurant, you look at a list of items, and you choose the thing that you like most. It might be hard to imagine anyone not knowing what a menu bar is nowadays, but when they were introduced, the metaphor meant that a user could quickly understand the purpose of them.

The advice in this section is based on common principles of interface design. Motion is a powerful tool for explaining the causes and effects of your actions, so be careful only to add animation that makes things clearer. In the following section, we will explore some examples.

Looking at focus

The first thing we're going to try is to guide the user's focus, to make it easier for them to see the information that they are interested in.

This covers two important principles: guiding the user's attention, and removing distractions. We will look at distractions first by examining a simple Application.

The Notification App is a mobile phone application for security guards who look after people's houses. It allows a security guard to electronically watch a set of houses from his or her phone. This allows the guard to pop out to the late night donut diner for a snack, while still being able to keep a watchful eye on the houses in his or her care.

The Notification App displays a list of all the houses that are being guarded. Each house is fitted with a sensor in the door that detects whenever someone comes into or out of the house. The application then uses this information whenever someone enters or leaves a building. If the security guard thinks that it might be a burglar, they will go and investigate.

The app displays security information in two ways, as follows:

- When someone enters or leaves, it pops up a notification dialog, in case you are watching the houses expecting a visitor
- The list of houses also contain information about the last event that happened, in case you need a quick overview of the most recent activity

While you are using the application, a security guard's torch moves from left to right at the bottom of the screen. The security guard torch is the logo for the company that makes the security system.

However, users are reporting that when there is a lot of activity in the application, the torch can be terribly distracting. We're going to test this out, and if it's true, we're going to get rid of it.

Time for action – don't confuse me with animation!

1. Import the project **Notifications** from the Code Bundle. Build it and deploy it to your device.
2. The application is currently set to "party simulation". Imagine that there is a street party going on, and people are going in and out of all of the houses in the street. In this sort of situation, your security guard will be seeing a lot of notifications.

From the point of view of our application, this means that our user interface will be displaying a lot of information. However, if you want to make the simulation even busier, you can open up `src/com/packt/animation/notifications/NotificationsView.java` and change the value of `NOTIFICATION_TIME` to something smaller.

The purpose of the party simulation is to challenge your observation skills while using the application; can you change how convenient it is to read by making a change to the way the user interface appears?

3. Launch the application. See how the data is coming from all four houses at once, and you are getting regular notifications at the same time.

You should try to read each and every notification for a while. Try saying the events out loud as they appear in shorthand, such as, "left Bob, left Amy, entered Steven". Get a feel for how comfortable the text is to read.

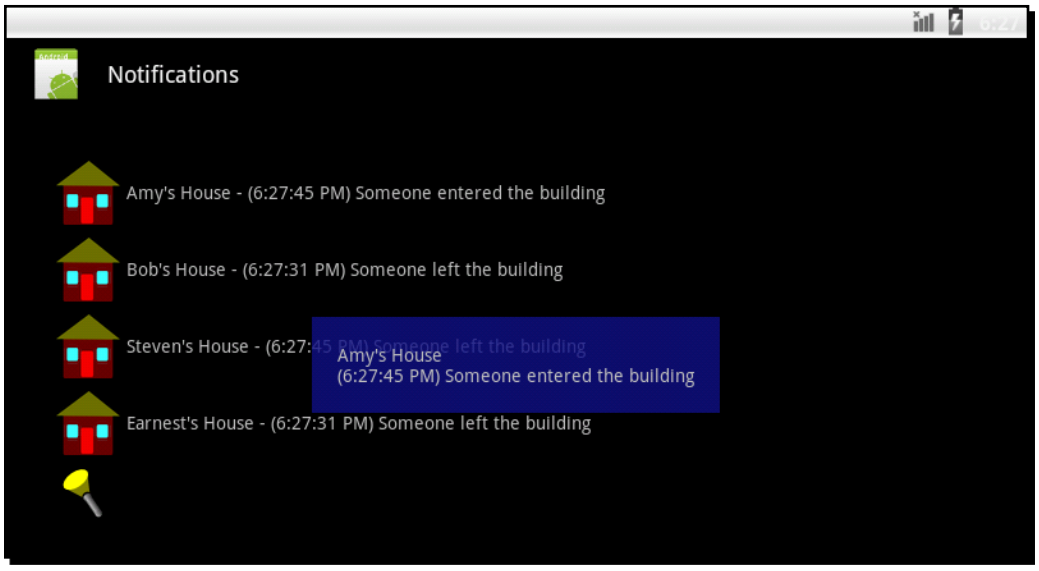
4. Now, we're going to try to remove the torch animation and see if that makes it any easier to use.

Navigate to `src/com/packt/animation/notifications/NotificationsActivity.java` and comment out the following lines:

```
//View torch = findViewById(R.id.torch);
//Animation torchAnim =
//  AnimationUtils.loadAnimation(
//      NotificationsActivity.this,
//      R.anim.torchanim);
//torch.startAnimation(torchAnim);
```

Here, we have removed the code that makes the torch move around the bottom of the screen.

5. Let's try it again! Build and run your new and improved version of the **Notifications** application.



When you build and run the application, the flashlight graphic should no longer be animated. We hope that this is going to make it easier to read the rapid-fire text from the party going on in the four houses.

6. Try reading the notifications again. Does it seem any easier?

   If you want to get a better feel for how the animation affects readability, go back and add in the lines that we commented out previously. How much difference does the animation make to the way that you personally read the notification information?

What just happened?

The torch animation was not displaying anything useful, but because it was animated, it became much more eye-catching than it needed to be. It was creating a focal point somewhere useless, and distracting the user from the important data.

When we removed the torch animation, the focal distraction was lost and the application became much more readable.

In order to investigate the problem, we created a pathological test case. We increased the speed of the application to a point that would only very rarely occur in an ordinary situation. In our hypothetical street party, the human eye struggles to keep track of how many people are entering and leaving the house. By making the application harder to use, we could highlight usability issues that might have otherwise been acceptable.

Everybody is different, so maybe you did not find the torch as distracting as I did. This is why, when doing usability tests, you should get at least five or six people to take a look at the application.

Have a go hero – usability testing

Grab some friends and colleagues, and ask them to try our application. Do the following usability test with them, where you will be writing and they will be talking:

1. Get a piece of paper, and write down "animated" and "no animation". This will be your scorecard for the two applications, so make sure there is enough space next to them to keep a running tally.
2. Show them the version of the application with the animation. Ask them to read out the house name, whenever a new notification pops up. Start a countdown timer for 1 minute's time.
3. Every time they stumble on a word, or get confused, put a mark next to "animation".
4. Show them the version of the application with no animation. Again they must read out the house name whenever a new notification pops up and starts a countdown for 1 minute.
5. Every time they stumble on a word, or get confused, put a mark next to "no animation".

How did the two applications fare?

A short disclaimer: this is not a tutorial on how to do usability testing; it's just a bit of fun. That said, the most important lesson in usability design is to get ordinary people to try your software as often as possible.

Getting to grips with metaphors

Next up, let's take a look at how we can use animation for creating user metaphors.

A metaphor, in this case, is a comparison to something in the real world. The two redeeming features of a user interface metaphor are as follows:

- The metaphor should be easily recognizable as something from the real world
- The metaphor should explain something by this comparison

In previous chapters, we have seen how easy it is to create an animation that moves from one place to another. One simple metaphor that we could make is a letter travelling from a house to the user.

- The metaphor is the postal system
- The metaphor explains that the message originates from that particular house

Let's see how this metaphor looks in the real world.

Time for action – getting messages from houses

Our users are happy with the changes that we made to reduce distraction, but now they are complaining that it is quite hard to see at a glance which message is coming from which house. They would like a visual cue that shows which house each notification is coming from.

To achieve this, we are going to add an animation that makes the notification fly out of its respective house, like a kind of letter. This is a simple metaphor and will help to visually associate the message with the house.

We are going to tone down "party mode" so that we can focus on making the notification animation clear and easy to understand.

1. First up, let's slow down the app a little bit. You can open up `src/com/packt/animation/notifications/NotificationsView.java` and change the value of `NOTIFICATION_TIME` to something longer, such as the following:

   ```
   public static final long NOTIFICATION_TIME = 3000;
   ```

   Here, we have set the time between showing notifications to 3000 milliseconds, which will give us a little more time to display an animation.

Please note that we're cheating a little bit here. We should really allow for the case when the notification activity is very fast, but for the sake of keeping this example simple, we can ignore that for now. Use your own ingenuity to solve that problem!

2. Next, we need to create a new animation for the pop-up notifications. This animation will need to be different depending on which house it comes from. So let us define it by creating a new method in `NotificationView.java`. First, we will need to add a few imports for the tween animation classes as follows:

   ```
   import android.view.animation.Animation;
   import android.view.animation.AnimationSet;
   import android.view.animation.ScaleAnimation;
   import android.view.animation.TranslateAnimation;
   ```

   These will take care of making our notifications zoom out from the houses.

3. Next, in the body of `NotificationView`, add the following new method:

```
   private Animation getNotificationAnimation (TextView houseView)
{
      AnimationSet mailAnimation = new AnimationSet(true);
      ScaleAnimation scale = new ScaleAnimation(0.1f, 1, 0.1f, 1);
      float letterboxX = houseView.getX()+27;
      float letterboxY = houseView.getY()+27;
      TranslateAnimation translate = new TranslateAnimation(
            Animation.ABSOLUTE, letterboxX - getX(),
            Animation.RELATIVE_TO_SELF, 0,
            Animation.ABSOLUTE, letterboxY - getY(),
            Animation.RELATIVE_TO_SELF, 0);
      mailAnimation.addAnimation(scale);
      mailAnimation.addAnimation(translate);
      mailAnimation.setDuration(1000);
      return mailAnimation;
}
```

This method assumes that it is receiving a house `TextView` as an argument. It adds a `ScaleAnimation`, and that is what gives the 3D feeling that the letter is flying from the background into the foreground.

We then get the coordinates of (approximately) the middle of the house graphic, so that the notification appears to be emitted from the door of the house (in Britain, the letterbox is in the middle of the door). We then translate the image from that coordinate into the default space in the middle of the screen.

4. We have created our new animation style; let us add it to our notifications.

Navigate to the anonymous `Runnable` in the middle of the `run` method in `NotificationView`. You can find the right one by looking for the comment that says: `// Here we display the NotificationView to the user.`

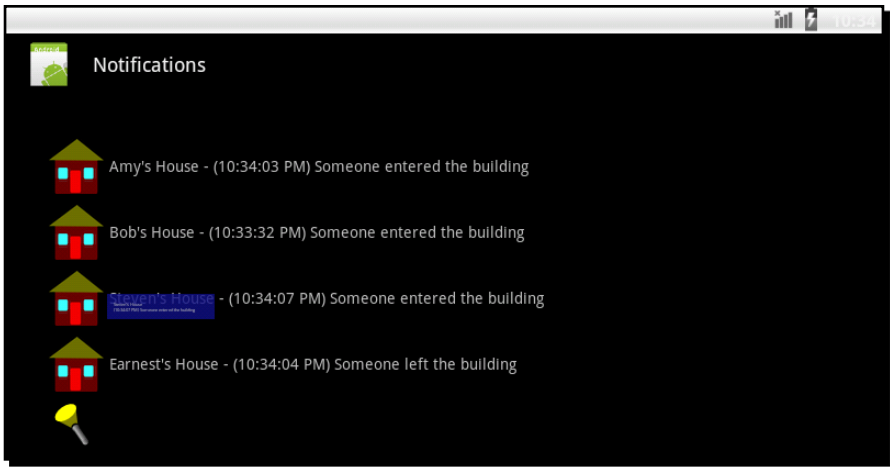
Update it to appear as follows (the highlighted bit is the only part that needs changing) :

```
               post( new Runnable()
{
               // Here we display the notification view to the
user.
               public void run()
{
                     currentNotification.ownerView.setText(
                        currentNotification.owner
                        + " - "
                        + currentNotification.message);
```

```
                        NotificationsView.this.setVisibility(
                           View.VISIBLE);
                        NotificationsView.this.startAnimation(
                           getNotificationAnimation(
                              currentNotification.ownerView));
                        NotificationsView.this.setText(
                           currentNotification.owner
                           + "\n"
                           + currentNotification.message);
}
});
```

5. Here, we have added a call to the new animation method. We pass it the view associated with the house, so that we know where to make the letter come out from.

6. Our animation is ready to test! Build and run the application. The following screenshot can be seen:



Here, you can see the notification originating from a house.

Do you think that the new animation makes it clearer where the notifications originate?

What just happened?

Here, we used an animated metaphor to give the user more information about what we are displaying on the screen. By understanding the metaphor of the sending of the message, the user can see at a glance which house is generating which notification.

We used a tween animation to make a notification appear very small in a particular portion of the screen, and then we brought it to the foreground by moving it to the center of the screen and making it bigger. We did this in a simple way by using a `ScaleAnimation` and a `TranslateAnimation`, tied together in an `AnimationSet`. The animation was calculated based on the location of the house view of the house that originated the message.

You do not need to understand much about how to use a computer in order to understand a metaphor like the one that we have created here, because it is based on concepts from everyday life. This sort of metaphor will make your application easier for users to pick up and understand for the first time.

Focus, redux

The metaphor also provides a visual guide to where we should be looking. Because the animation is deliberately associated with a particular house, the user's focus is directed toward that house.

Unlike the distracting torch animation from our last example, this is a positive example of using animation to provide focus.

Maintaining consistency within an application

When you are creating metaphorical descriptions, try to use the same metaphor for everything that behaves in a similar way. Users learn the way that one thing works, and then they try to apply that knowledge to other parts of the application.

If, for instance, the houses all had different messaging animations, your poor users would spend a lot of time wondering why one house is different from another. What caused that inconsistency? Don't do it just because you think it might look prettier.

Usability testing is a great way to find inconsistent metaphors. Users will ask great questions such as, "if that animation shows my parcel being flown in by plane, why do I see the same animation if it's delivered by hand?" They will probably spot things that you just thought were completely inconsequential.

Pop quiz – usability

1. What is a good thing to have in a user metaphor?
    a. Color
    b. Speed
    c. Consistency

2. Why would you want to use animation to provide focus?
    a. To make the application prettier
    b. To guide a user to where they should be looking
    c. To distract users
3. How can unnecessary animation affect your application?
    a. It wears out the screen
    b. It makes your eyes blurry
    c. It's distracting

Have a go hero – more usability testing

Grab some friends and colleagues and get them to try out the old and the new applications.

Give them a simple task to do using the application. For instance, you could ask them to touch the house that originated the current notification. See if the visual cue of our new animation is enough to make it clear where the notification messages are coming from, or whether the user is happy just reading the house name from the top of the notification message.

Like before, you may want to look for measurable cues like mistakes or hesitation, or you may simply ask them which application feels more useful.

Reducing power usage

Your average Android device will spend most of its life on battery power. Unlike a desktop computer, which is plugged into a power source a lot of the time, if your Android application uses a lot of processor time or graphics processing, your user will notice the difference. If they install your application on their mobile phone or tablet, and suddenly the battery life decreases dramatically, they will do two things. Firstly, they will uninstall your app (which is not good). Secondly, they will hate you forever.

This section is most useful with animations that are based on surfaces, although all animations that require a lot of operations will benefit from testing. For example, if you use a lot of custom `ValueAnimators` to update lots of different views.

I'm not sure if I could live with the idea that my users hated me forever, so we must address these issues! Before we begin to see how we can make our animations more power-efficient, let us see how we can identify if our application is using a lot of power.

You may be aware of the Battery Use screen on Android. It lives in the following:

Settings | **About Phone** | **Battery Use**

Using this tool, you can see how much power your application is using compared to other applications. This can be handy if you do not have any better tools at hand, but there is a much better tool available, known as **PowerTutor**.

Let me introduce PowerTutor. PowerTutor was written by Mark Gordon, Lide Zhang, and Birjodh Tiwana of the University of Michigan.

It is a detailed statistics-gathering and presentation application based around resource usage on Android devices. Rather than the built-in **Battery Usage** screen, this application yields much more information that is useful to application development.

Visit their site at `http://powertutor.org` for more information and to learn more useful applications of the tool.

With it, we can navigate to the statistics related to our application, and see how much power we are using.

Let's get started with a simple example.

Time for action – measuring battery usage with PowerTutor

1. The first thing we need to do is get PowerTutor. You can download it from the Android Market or get it from `http://powertutor.org` and install it to your Android device.

   PowerTutor will only work with real Android devices. You cannot use an emulator for this test.

2. Open up PowerTutor on your device. When you first open it up, you will need to accept an EULA. Read through it, and if you are satisfied, you can accept it to continue.

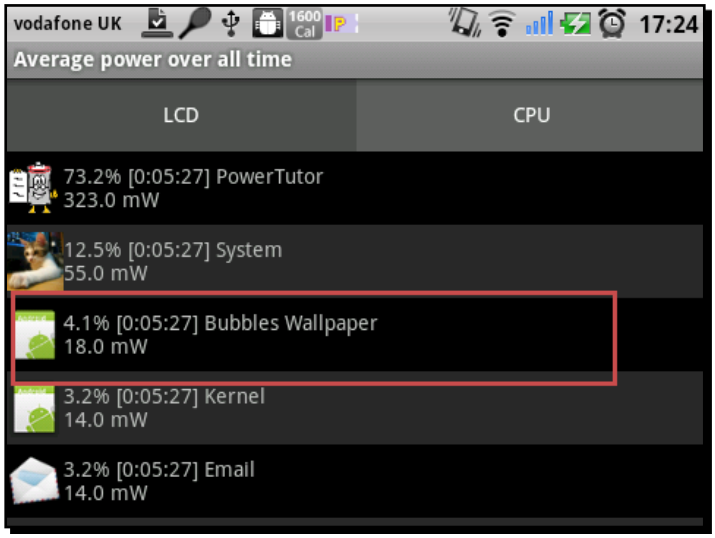
   You will notice here that the PowerTutor app is only supported on some devices. If you use an unsupported device, don't despair! The figures are still useful, as they will help you understand how much power your animation is using relative to other applications.

3. Hit **Start Profiler** to begin logging power usage.

4. Navigate away from the PowerTutor application, and try running another application for a minute or so.

5. Return to the PowerTutor Application and push the **Application Viewer** button to view the power consumed by each application. If you see a set of graphs instead of a list of buttons, hit the **back** button and then press the **Application Viewer** button on the screen that appears.

6. You will see a row of buttons at the top of the screen that corresponds to things on the phone that may use power. Use your common sense when choosing these. We will generally be interested in the **LCD** or **OLED** values (depending on your handset), and the **CPU**. Touch them to illuminate them, and gray out any others.



Your device may appear slightly differently to this one, as the PowerTutor provides extra monitoring capabilities where available.

7. Look for the application you were using while the PowerTutor profiler was running. You will see that it has an entry that looks a little like the following:

   6.1% [0:33:21] Test Application

   169.0 mW

   The value in mW (milliwatts) is the most interesting for us, as it is an estimated value of how much juice our application needs!

8. We are now done with the profiling behavior in PowerTutor. You can disable it by pressing the **back** button on your device, and picking the **Stop Profiler** option.

What just happened?

By using PowerTutor, we could get a useful figure for how much power an application on the target device was using.

It helps to run the application under test for a few minutes before reading a measurement from PowerTutor, to give us a better idea of the average power consumption. Also, PowerTutor will measure the power usage of an Activity regardless of whether it is on the screen. Therefore, to measure something in an active state (like being on screen), make sure that you do not let it stay idle too long before taking a measurement. This is not a rigidly accurate approach, but it is still useful to our needs.

Precise estimation

PowerTutor does not support all devices, so we cannot guarantee that the figures are accurate. But we can use them for relative comparisons between applications. Moreover, by using it regularly to test your animations, you can ensure that they are not using an excessive amount of memory.

Changing the Application Viewer Timespan

Often, you will want to disregard the previous behavior of an application and look at only the last minute or so. Especially, if it has been idling and perhaps not running the animation code that you want to test.

From the **Application Viewer** pane, if you press the **Menu** button on your device, you will see that there is a **Time Span** option on the menu. This allows you to control what duration you are sampling data from, from a minute to a day.

PowerTutor-supported devices

To discover which devices are best supported by PowerTutor, visit their website at `http://powertutor.org`

To do the exercises that are given in this book, you do not necessarily need to be using a supported device.

What is a milliwatt?

A milliwatt is one thousandth of a Watt. A watt is a measurement of electrical **power**- the amount of energy used per second. Because a Watt is a unit that takes time into account, the length of time we spend measuring should not affect the reading that PowerTutor gives us, unless our application behavior changes.

A battery is of course a store of energy. The overall power usage of our device determines how long the device will keep working on one charge of the battery. If our application uses less power (fewer milliwatts), then the battery will last a little bit longer, and our user will be a little bit happier! Now, we know how to tell how much energy our application is taking, how does this apply to animation? I'm glad you asked that, because that's what the next section is about.

Optimizing an animation for power

We will do this in a few iterations, so I'll ask you to refer to the next subsection every time you need to measure the power usage of the application, you will need to use the previous PowerTutor exercise. Remember to let the application run for a few minutes to get a good power estimate.

Looking for problems

The first thing we should do with an animation is to check that it is behaving responsibly with respect to its power consumption. For this, we can use PowerTutor to see how it is behaving on the device.

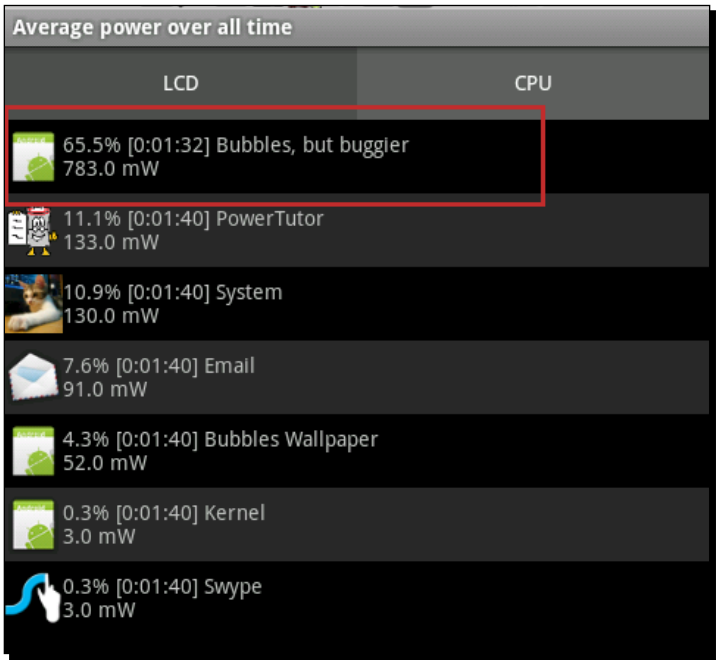
When checking an animation, you should create a separate test Activity for it, so that other functionality in your code does not affect the power measurement. This way, by measuring the power usage, you can be confident that you are looking at the animation itself.

Time for action – identifying a problem

Oh boy, we have a bad problem with our `BubblesView` class. Some nasty `gremlin` has got into the system and is making it really really inefficient. What can we do? We must investigate as follows:

1. Get the `BuggyBubbles` application from the code bundle. This is a simple Activity that displays a `BubblesView` animation. The activity doesn't do anything else, so we can be confident that we are primarily measuring the animation behavior.
2. Build the `BuggyBubbles` activity and send it to your device.

3. Go in to PowerTutor on your device and start profiling.
4. Launch the **Bubbles, but buggier** application and let it run for a minute.
5. Return to PowerTutor and use the **Application Viewer** to see how much power it is using.



What just happened?

We just used PowerTutor to usefully measure the power consumed by our animation.

As you can see, it's using a lot more power than any of the other applications on the list. On my device, it comes up as 783 milliwatts. This is clearly unacceptable for a simple animation. We now know that we should investigate further!

Finding the power hogs

Now, we will need to delve into our animation to see exactly why it is using so much power. This is a good opportunity to use one of the Android Development Kit tools, called **Traceview**.

To use **Traceview**, we need to go in to the code of our animation and tell it to create a trace file. Then, while the trace is running, a file will be created on our Android device. We can use that file to see what methods are running the most, and therefore where we should optimize.

Let's try it; you'll see what I mean.

Time for action – tracing to find optimizations

So far, we have identified a problem: our animation is really power hungry. This is not good. We must find out where the program is doing work that it doesn't need to. The steps for it are as follows:

1. In the `BuggyBubbles` project, navigate to the file `src/com/packt/animation/buggybubbles/BuggyBubblesActivity.java` and add the following new import statement:

   ```
   import android.os.Debug;
   ```

   I'm sure it will come as no surprise to learn that this class provides debugging capabilities.

2. Next up, add a new line to the `onCreate` method in `BuggyBubblesActivity`, such as the following:

   ```
       public void onCreate(Bundle savedInstanceState)
   {
           super.onCreate(savedInstanceState);
           setContentView(R.layout.main);
           Debug.startMethodTracing("buggybubbles");
   }
   ```

   When the application is launched, this will create a new file in your device's removable storage, usually in `/sdcard/`. The file will be called `buggybubbles.trace`.

3. We need to ensure that our application does not write out trace information, when it doesn't need to. So inside `BuggyBubblesActivity`, add the following new method:

   ```
       public void onPause()
   {
         super.onPause();
         Debug.stopMethodTracing();
   }
   ```

   This will stop the trace file from being filled up too soon.

4. Compile the application and deploy it to your device. You will see the bubbles activity run.

5. Let the bubbles run for a few seconds, and then leave the activity so that debugging will stop.

6. Pull the file `buggybubbles.trace` from your device's removable storage to your computer. You can do this from the File Explorer in the DDMS view for Eclipse, or you can use `adb` from the following command line:

   ```
   adb pull /sdcard/buggybubbles.trace.
   ```

7. Using a terminal on Linux or Mac, or `cmd.exe` on Windows, navigate to the directory where you copied the `buggybubbles.trace` file on your local computer.

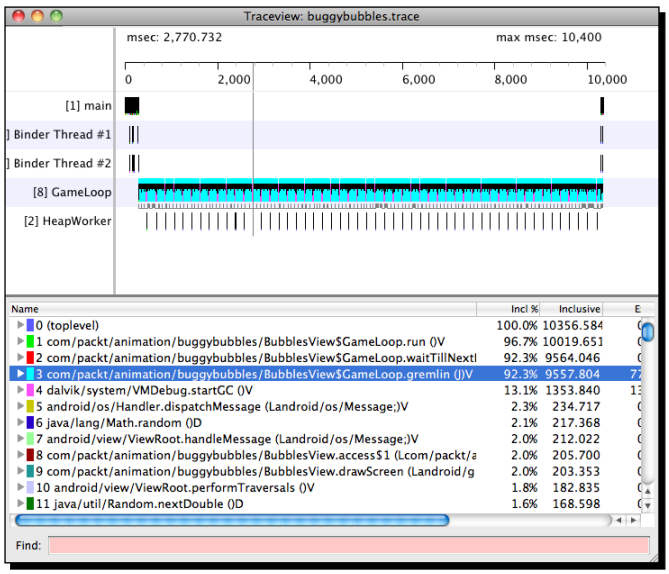
   This step assumes that you have your Android tools directory in your path. If you do not, please read the Android documentation to find out how to do this. You can find the relevant setup document here: `http://developer.android.com/sdk/installing.html#sdkContents`

   Once your path is set up, type the following line:

   ```
   traceview buggybubbles.trace
   ```

   Windows users should note that, instead of typing `buggybubbles.trace` on its own, you will have to pass in the full path to where you copied the trace file. For example, `traceview C:\Documents\BuggyBubbles\buggybubbles.trace`

   This will bring up a screen that looks like the following one:



There are two parts of the **Traceview** window, a top half and a bottom half.

8. The top half shows a timeline of all of the threads in your application. Time runs from left to right, from the time that you started the trace file on the left, to the time that you exited it on the right. The horizontal stripy bars are threads in your application. Move your mouse over the bars in the top half; you will see the methods that were running in those threads at that time.

9. Try zooming in on a particular part of the timeline by clicking and dragging from left to right on a timeline segment. This will select a chunk of the timeline and show it in a bit more detail. This is helpful if the timeline view is too crowded with information to read easily.

10. The bottom half of the **Traceview** window shows a list of methods in your application, sorted by how often they get used. Try clicking on them; they expand out to give you more information about the method and its relationship to the application as a whole.

11. Have a look around the trace generated by your application. There is one method that has a suspicious name, and takes an awful lot of processing time. See if you can find it in both panes (this should be easy).

    Read off the exact Java qualified name of the `gremlin` method. We will use this in the next section.

What just happened?

Here, we saw how **Traceview** can be used to browse through the activity in our animation. What is particularly useful is the way that it shows methods that spend a lot of time running.

When a method is running, it is using CPU time, and therefore it is using up battery energy. To use less power, reduce the amount of time your methods spend running.

You should have had no difficulty at all finding the offending `gremlin` method. In the top pane, the `gremlin` will have occupied most of the `GameLoop` thread, taking over the entire system for its processing requirements. No wonder then that it was causing so much power consumption in PowerTutor.

In the lower pane, the `gremlin` method would be in the top five methods sorted by processor time.

Removing the gremlin

Now that you know where the `gremlin` method is, let us navigate to it and fix it.

Time for action – squashing gremlins that use too much power

1. Open up `src/com/packt/animation/buggybubbles/BubblesView.java`, the Java file that contains the `gremlin` method, and take a look at the contents of the method that are as follows:

```
      private void gremlin(long nextSleep)
{
         while (nextSleep>0)
{
            double[] pointlessMemoryAllocation = new double[1337];
            double pointlessCalculationValue =
               Math.random()*66666.66;
            for (
               int i = 0;
               i< pointlessMemoryAllocation.length;
                 ++i)
{
                  pointlessMemoryAllocation[i] =
                    pointlessCalculationValue / (1+i);
}
            nextSleep = frameTime - System.currentTimeMillis();
}
}
```

Yuck! That really is a nasty method. It looks like the `gremlin` has replaced our call to `Thread.sleep` with a piece of code that just does some meaningless processing until the sleep has come to an end.

Find where this method is called. It should be in `waitTillNextFrame` as follows:

```
   private void waitTillNextFrame()
{
         long nextSleep = 0;
         frameTime += msPerFrame;
         nextSleep = frameTime - System.currentTimeMillis();
         // HAHA THEY WILL NEVER FIND ME!
         gremlin(nextSleep);

         /*
         if (nextSleep > 0)
{
```

```
            try
{
               sleep(nextSleep);
}
catch (InterruptedException e)
{
}
}*/
}
```

The `gremlin` is laughing at us! But we've found it.

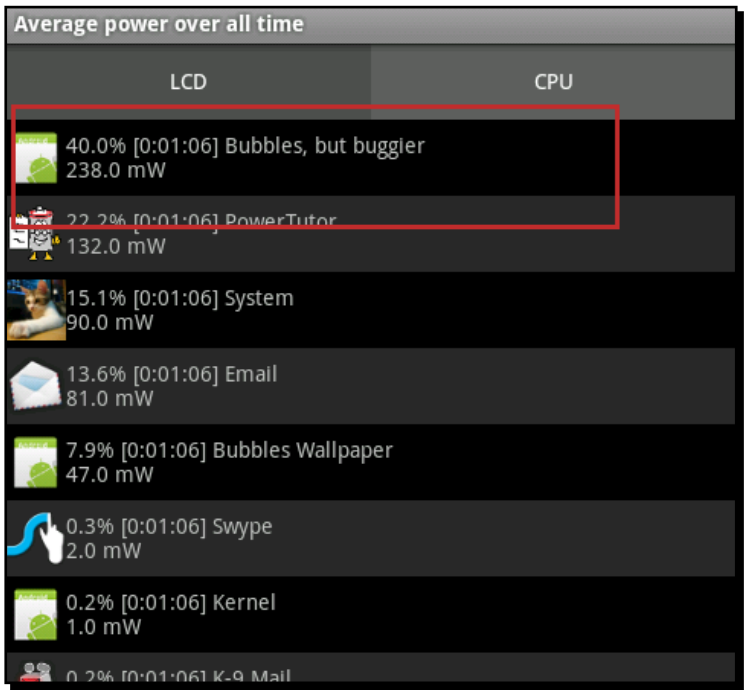
2. Time now to delete the `gremlin` code and reinstate our old `sleep` method. In `waitTillNextFrame`, remove the call to `gremlin` and uncomment the use of the `sleep` method.

```
      private void waitTillNextFrame()
{
         long nextSleep = 0;
         frameTime += msPerFrame;
         nextSleep = frameTime - System.currentTimeMillis();

         if (nextSleep > 0)
{
            try
{
               sleep(nextSleep);
}
catch (InterruptedException e)
{
}
}
}
```

There, we should be using `Thread.sleep` for timing now. This means that we are not draining power by over-working the CPU.

3. To tidy up, delete the method `gremlin`.
4. Build and deploy `BuggyBubbles` to your device.
5. Use PowerTutor to start logging power usage.
6. Start running `BuggyBubbles` for a bit.

7. Go back to PowerTutor and read off the power consumption.



8. Finally, remove the calls to create logs. We do not want to be filling up the removable storage on your device when we don't need to.

Open up `BuggyBubblesActivity.java` again and strip out the lines highlighted as follows:

```
public class BuggyBubblesActivity extends Activity
{
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState)
{
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
     Debug.startMethodTracing("buggybubbles");
}

 @Override
 public void onPause() {
       super.onPause();
```

```
        Debug.stopMethodTracing();
    }
}
```

This will keep our application from writing out unnecessary trace files.

Of course, if you have purposefully created an Activity just to test your animation's performance, you may wish to leave this instrumentation in for later use.

What just happened?

We used the information that we had gained from PowerTutor and **Traceview** to locate a performance issue.

When we navigated to the code in question, it was very easy to see that there was a lot of work being done for no reason, and that we could remove the redundant code.

When you are using these tools to optimize your own code, the cause of the bottlenecks may not be so obvious. It does provide a good starting point for your own optimization, and the following resources may help you to optimize your application further.

Once we had optimized the code, we went back and tested that the optimizations had indeed improved our power usage.

Optimizing using an easy recipe

The way we used the tools at our disposal forms a simple script to optimize an animation, or indeed any Android application that needs a performance boost.

The basic recipe for optimization for power usage goes something as follows:

1. Create a test activity in your project that simply displays the animation
    - In this case, we had pre-prepared an activity that uses the `BubblesView` class.
2. Identify if your animation needs optimization
    - In our example, we used PowerTutor to read the power usage of the application.
3. Profile your test activity to find power-hungry methods
    - We used the **Traceview** application that comes with the Android Development Kit to search for the most CPU-hungry methods.

4. Examine the methods for optimizations
    - Simply by looking at the `gremlin` method, we saw that it was an inefficient way to add time looping to the `GameLoop` thread. Other optimizations may be less obvious.
5. Apply any optimizations you can find
    - In the previous example, we stripped out the `gremlin` code and replaced it with a simple `Thread.sleep`.
6. Test to see if the optimizations have produced a performance boost
    - To do this, we returned to PowerTutor and profiled the application again.

Once you have been through the previous process, you can go back to developing your application and use the animation again, safe in the knowledge that it is not causing a power drain for your users.

The Android development team put together a handy document on how to optimize your code. It states that you should not use CPU resources frivolously, and you should be aware that Java memory allocations incur performance costs.

It then goes on to explain some helpful techniques for optimizing your code.

There is far too much information in that document to properly do it justice here, but you should check it out at the following address:

`http://developer.android.com/guide/practices/design/performance.html`

Less is sometimes more

Of course, the most effective way to reduce CPU usage is to do fewer things. If you can reduce the number of moving elements in your animation, you can reduce the power usage accordingly.

Think of the Bubbles Wallpaper application, for instance. Every bubble requires a calculation at every frame. If we reduced the number of frames, and subsequently the number of calculations that the animation requires, then we reduce the power consumption.

Pop quiz – power usage

1. Why would you want to reduce the power consumption of your application?
    a. It wears out the CPU
    b. It drains battery life
    c. It makes the screen dim
2. What is the name of the Android tool for looking at thread activity?
    a. stacktrace
    b. showthreads
    c. traceview
3. What is the name of the tool that we used to measure power consumption?
    a. PowerMetal
    b. MichiganTutor
    c. PowerTutor
4. What is the unit of electrical power?
    a. Watt
    b. Joule
    c. That is a terrible joke

Have a go hero – using the Android performance guidelines

Open up the Android performance guidelines from the following:

`http://developer.android.com/guide/practices/design/performance.html`

Have a read through and try to understand as many of the points that it makes as you can.

Open up the `BubblesView` class, and navigate to the `calculateDisplay` section of the game loop. Can you find any optimizations based on the information in the Android document? Do you think that they'll make a big difference to performance in this case?

Summary

In this chapter, we took a look at some ways to make our animations better for the end user. This takes two major approaches: usability and efficiency.

Specifically, we covered the following:

- How to direct a user's attention using animation
- How a bad animation can distract a user
- Using a consistent metaphor to explain behavior to a user
- Measuring power usage of an animation
- Analyzing your animation as it runs through time
- Spotting things to optimize

The purpose of this book has been to learn how to create great animations. The previous chapters have been primarily about how to create animations, and this chapter hopefully gave you some insight into how to make them great.

I hope that this book has been interesting and helpful to you. I wish you all the best when creating your next animated masterpiece!

Pop Quiz Answers

Chapter 1: Animation Techniques on Android

View animations and Drawable animations

1	b
2	a
3	c
4	c
5	a

Putting it all together

1	b
2	c
3	a
4	a
5	c
6	c

Chapter 2: Frame Animations

Making frame animations

1	c
2	b
3	b
4	a
5	a

Controlling frame animations

1	a
2	c
3	c

Transition Drawables

1	c
2	a
3	c
4	b

Chapter 3: Tweening and Using Animators

All those tweens!

1	c
2	b
3	d
4	b
5	a

AnimationListeners

1	b
2	b

Interpolators

1	c
2	a
3	c

Chapter 4: Animating Properties and Tweening Pages

ViewFlippers

1	b
2	a
3	b

Java tweens

1	b
2	c

ObjectAnimators

1	b
2	a

Value Animators

1	a
2	b

Chapter 5: Creating Classes for Tween Animation

PropertyValueHolders, ObjectAnimators, and TypeEvaluators

1	c
2	a
3	b
4	b
5	a

Fragment Animation and XML Animators

1	c
2	c
3	c

Custom interpolators

1	b
2	c

Chapter 6: Using 3D Visual Techniques

Depth effects

1	b
2	c

3D rotations

1	b
2	a
3	a,b

Chapter 7: 2D Graphics with Surfaces

Surface animations

1	c
2	a
3	d
4	c
5	a
6	c

Chapter 8: Live Wallpapers

Live wallpapers

1	a
2	b
3	c
4	b

Interactivity

1	b
2	a
3	c

Preferences for live wallpapers

1	c
2	a
3	b

Chapter 9: Practicing Good Practice and Style

Usability

1	c
2	b
3	c

Power usage

1	b
2	c
3	c
4	a

Index

Symbols

A

B

C

D

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Thank you for buying
Android 3.0 Animations: Beginner's Guide

About Packt Publishing

Packt, pronounced 'packed', published its first book "Mastering phpMyAdmin for Effective MySQL Management" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

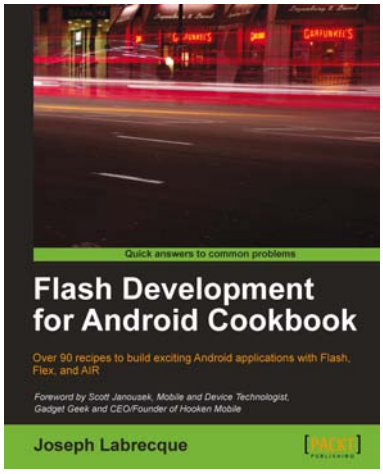
Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.PacktPub.com`.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

[PACKT]
PUBLISHING



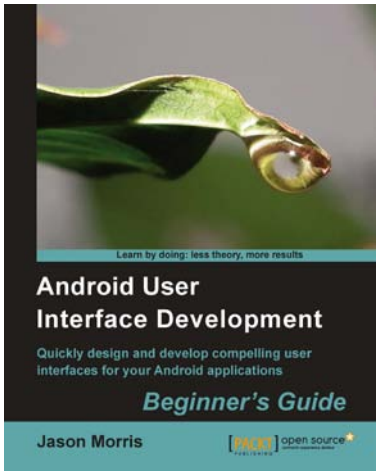
Flash Development for Android Cookbook

ISBN: 978-1-84969-142-0 Paperback:372 pages

Over 90 recipes to build exciting Android applications with Flash, Flex, and AIR

1. The quickest way to solve your problems with building Flash applications for Android
2. Contains a variety of recipes to demonstrate mobile Android concepts and provide a solid foundation for your ideas to grow
3. Learn from a practical set of examples how to take advantage of multitouch, geolocation, the accelerometer, and more
4. Optimize and configure your application for worldwide distribution through the Android Market

---



Android User Interface Development

ISBN: 978-1-84951-448-4 Paperback: 304 pages

Quickly design and develop compelling user interfaces for your Android applications

1. Leverage the Android platform's flexibility and power to design impactful user-interfaces
2. Build compelling, user-friendly applications that will look great on any Android device
3. Make your application stand out from the rest with styles and themes

[PACKT]
PUBLISHING

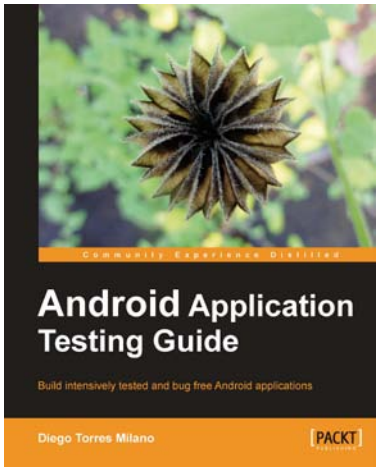
Android 3.0 Application Development Cookbook

ISBN: 978-1-84951-294-7 Paperback:272 pages

Over 70 working recipes covering every aspect of Android development

1. Written for Android 3.0 but also applicable to lower versions
2. Quickly develop applications that take advantage of the very latest mobile technologies, including web apps, sensors, and touch screens
3. Part of Packt's Cookbook series: Discover tips and tricks for varied and imaginative uses of the latest Android features

---



Android Application Testing Guide

ISBN: 978-1-84951-350-0 Paperback: 332 pages

Build intensively tested and bug free Android applications

1. The first and only book that focuses on testing Android applications
2. Step-by-step approach clearly explaining the most efficient testing methodologies
3. Real world examples with practical test cases that you can reuse