IMPLEMENTING MATERIAL DESIGN
FOR DEVELOPERS

ANDROID USER INTERFACE DESIGN

SECOND EDITION

Ian G. **CLIFTON**

Android™ User Interface Design

Addison-Wesley Usability and HCI Series

Visit **informit.com/series/usability** for a complete list of available publications.

Essential Guides for Human-Computer Interaction and User Interface Designers

Books in the HCI and Usability series provide practicing programmers with unique, high-quality references and tutorials on interaction and interface design, a critical component of success for any mobile app or website. The books in this series bring the full range of methods and options available to meet the challenge of designing for a natural and intuitive global user experience.

Make sure to connect with us!
informit.com/socialconnect

informIT.com the trusted technology learning source | Addison-Wesley | Safari

Android™ User Interface Design

Implementing Material Design for Developers

Second Edition

Ian G. Clifton

Addison-Wesley

New York • Boston • Indianapolis • San Francisco
Toronto • Montreal • London • Munich • Paris • Madrid
Cape Town • Sydney • Tokyo • Singapore • Mexico City

Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2015950113

ISBN-13: 978-0-134-19140-9
ISBN-10: 0-134-19140-4

Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.

First printing: November 2015

Editor-in-Chief
Mark Taub

Executive Editor
Laura Lewin

Development Editor
Songlin Qiu

Managing Editor
Kristy Hart

Project Editor
Namita Gahtori

Copy Editor
Cenveo® Publisher Services

Indexer
Cenveo Publisher Services

Proofreader
Cenveo Publisher Services

Technical Reviewers
Cameron Banga
Joshua Jamison
Adam Porter

Editorial Assistant
Olivia Basegio

Cover Designer
Chuti Prastersith

Compositor
Cenveo Publisher Services

Dedicated to those who care about user experience

This page intentionally left blank

Contents at a Glance

Introduction 1

Part I **The Basics of Android User Interfaces**

1 Android UI and Material Design. 5

2 Understanding Views—The UI Building Blocks 27

3 Creating Full Layouts With View Groups and Fragments 51

4 Adding App Graphics and Resources 75

Part II **The Full Design and Development Process**

5 Starting A New App 107

6 Prototyping and Developing the App Foundation. 129

7 Designing the Visuals 167

8 Applying the Design. 191

9 Polishing with Animations 231

Part III **Advanced Topics for Android User Interfaces**

10 Using Advanced Techniques. 263

11 Working with the Canvas and Advanced Drawing 303

12 Developing Custom Views 331

13 Handling Input and Scrolling 349

Appendix A Google Play Assets 387

Appendix B Common Task Reference 403

Index. 417

This page intentionally left blank

Contents

Introduction 1

Audience for This Book 1
Organization of This Book 1
How to Use This Book 2
This Book's Website 2
Conventions Used in This Book 2

Part I The Basics of Android User Interfaces

1 Android UI and Material Design 5

A Brief History of Android Design 6
Material Design 8
The Android Design Website 13
Core Principles 14
Standard Components 20
Supporting Multiple Devices 23
Avoiding Painful Mistakes 24
Summary 26

2 Understanding Views—The UI Building Blocks 27

What Is a View? 28
Displaying Text 34
Displaying Images 39
Views for Gathering User Input 42
Other Notable Views 45
Listening to Events 47
Other Listeners 48
Summary 49

3 Creating Full Layouts With View Groups and Fragments 51

Understanding ViewGroup and the Common Implementations 52

Encapsulating View Logic with Fragments 64

The Support Library 70

Summary 74

4 Adding App Graphics and Resources 75

Introduction to Resources in Android 76

Resource Qualifiers 76

Understanding Density 81

Supported Image Files 83

Nine-Patch Images 84

XML Drawables 86

Other Resources 101

Summary 106

Part II The Full Design and Development Process

5 Starting A New App 107

Design Methods 108

Defining Goals 110

High-Level Flow 114

Wireframes 117

Continuing with Content Pieces 122

Summary 128

6 Prototyping and Developing the App Foundation 129

Organizing into Activities and Fragments 130

Creating the First Prototype 131

Evaluating the First Prototype 160

Summary 165

7 Designing the Visuals 167

Wireframes and Graphical Design 168
Tools 168
Styles 169
Lighting 171
Colors 171
Text Considerations 178
Other Considerations 183
Designing Step-by-Step 185
Summary 189

8 Applying the Design 191

Working with the Designer 192
Slicing the Graphics Assets 193
Themes and Styles 199
Breaking Comps into Views 200
Developing the Woodworking App 201
Basic Testing Across Device Types 228
Summary 229

9 Polishing with Animations 231

Purpose of Animations 232
View Animations 232
Property Animations 233
Property Animation Control 235
ViewPropertyAnimator 241
Animating Form Errors 242
Animating Icons 246
Simple Transitions 252
Summary 262

Part III Advanced Topics for Android User Interfaces

10 Using Advanced Techniques 263

Identifying Jank 264
Using Systrace to Understand Jank. 265
Optimizing Images 272
Additional Performance Improvements. 283
Hierarchy Viewer 289
Custom Fonts 293
Complex TextViews 295
RecyclerView. 299
Summary 300

11 Working with the Canvas and Advanced Drawing 303

Creating Custom Drawables 304
Paint 305
Canvas. 305
Working with Text 306
Working with Images 310
Color Filters 313
Shaders 325
Summary 330

12 Developing Custom Views 331

General Concepts. 332
Measurement 332
Layout 333
Drawing. 333
Saving and Restoring State. 334
Creating a Custom View. 334
Summary 347

13 Handling Input and Scrolling 349

Touch Input 350
Other Forms of Input 351
Creating a Custom View. 352
Summary 385

Appendix A Google Play Assets 387

Application Description. 388
The Change Log. 389
Application Icon. 389
Screenshots 395
Feature Graphic 397
Promotional Graphic. 399
Video (YouTube). 400
Promoting Your App 400
Amazon Appstore 401

Appendix B Common Task Reference 403

Dismissing the Software Keyboard. 404
Using Full Screen Mode 404
Keeping the Screen On 405
Determining the Device's Physical Screen Size 406
Determining the Device's Screen Size in Pixels 406
Determining the Device DPI 407
Checking for a Network Connection. 408
Checking if the Current Thread Is the UI Thread 408
Custom View Attributes. 409

Index. 417

This page intentionally left blank

PREFACE

Android has evolved at an incredible speed, and keeping up with the changes is a difficult job for any developer. While working to keep up with the latest features and API changes, it can be easy to neglect the design changes Android is undergoing. When Google announced the Material Design guidelines, even designers who had long dismissed Android's visuals started paying attention.

It's more important than ever for Android developers to understand the core aspects of design and the Material Design guidelines go some of the way toward making that possible; however, without years of background in design, it can be difficult to make sense of everything. This book will guide you through the real-world process of design starting from an abstract idea and sketches on paper and working all the way through animations, RenderScript, and custom views. The idea is to touch on each of the core concepts and cover enough so that you can have productive conversations with designers or even create everything yourself.

Design has many purposes, but two of the most important are usability and visual appeal. You want brand-new users to be able to jump into your app and get started without any effort because mobile users are more impatient than users of nearly any other platform. Users need to know exactly what they can interact with, and they need to be able to do so in a hurry while distracted. That also means you have to be mindful of what platform conventions are in order to take advantage of learned behavior.

If you have picked up this book, I probably do not need to go on and on about how important design is. You get it. You want to make the commitment of making beautiful apps that are a pleasure to use.

This book will serve as a tutorial for the entire design and implementation process as well as a handy reference that you can keep using again and again. You will understand how to talk with designers and developers alike to make the best applications possible. You will be able to make apps that are visually appealing while still easy to change when those last-minute design requests inevitably come in.

Ultimately, designers and developers both want their apps to be amazing, and I am excited to teach you how to make that happen.

—Ian G. Clifton

ACKNOWLEDGMENTS

You would think that the second edition of a book would be easier than the first, but when you find yourself rewriting 90 percent of it because both the technology and design trends are changing so rapidly, it helps to have assistance. Executive Editor, Laura Lewin, once again helped keep me on track even as I restructured the book and dove in depth in places I didn't originally expect. Olivia Basegio, the Editorial Assistant, kept track of all the moving pieces, including getting the Rough Cuts online so that interested readers could get a glimpse into the book as it evolved. Songlin Qiu was the Development Editor again and took on the task of making sense of my late-night draft chapters. I am also extremely appreciative of the work done by the technical reviewers, Adam Porter, Cameron Banga, and Joshua Jamison, whose feedback was instrumental in the quality of this book.

ABOUT THE AUTHOR

Ian G. Clifton is a professional Android application developer, user experience advocate, and author. He has worked with many developers and designers, and led Android teams, creating well-known apps such as Saga, CNET News, CBS News, and more.

Ian's love of technology, art, and user experience has led him along a variety of paths. In addition to Android development, he has done platform, web, and desktop development. He served in the U.S. Air Force as a Satellite, Wideband, and Telemetry Systems Journeyman and has also created quite a bit of art with pencil, charcoal, brush, camera, and even wood.

You can follow Ian G. Clifton on Twitter at http://twitter.com/IanGClifton and see his thoughts about mobile development on his blog at http://blog.iangclifton.com. He also published a video series called "The Essentials of Android Application Development LiveLessons, 2nd Edition," available at http://goo.gl/4jr2j0.

This page intentionally left blank

INTRODUCTION

Audience for This Book

This book is intended primarily for Android developers who want to better understand user interfaces (UI) in Android. To focus on the important topics of Android UI design, this book makes the assumption that you already have a basic understanding of Android, so if you haven't made a "Hello, World" Android app or set up your computer for development, you should do so before reading this book (the Android developer site is a good place to start: http://developer.android.com/training/basics/firstapp/index.html).

Most developers have limited or no design experience, so this book makes no assumptions that you understand design. Whenever a design topic is important, such as choosing colors, this book will walk you through the basics, so that you can feel confident making your own decisions and understand what goes into those decisions.

Organization of This Book

This book is organized into a few parts. Part I, "The Basics of Android User Interface," provides an overview of the Android UI and trends before diving into the specific classes used to create an interface in Android. It also covers the use of graphics and resources. Part II, "The Full Design and Development Process," mirrors the stages of app development, starting with just ideas and goals, working through wireframes and prototypes, and developing complete apps that include efficient layouts, animations, and more. Part III, "Advanced Topics for Android User Interfaces," explores much more complex topics including troubleshooting UI performance problems with Systrace and creating custom views that handle drawing, scrolling, and state saving.

This book also has two appendices. The first focuses on Google Play assets (and covers the differences to know about when preparing for the Amazon Appstore as well), diving into app icon creation. The second covers a variety of common UI-related tasks that are good to know but don't necessarily fit elsewhere (such as custom view attributes).

The emphasis throughout is on implementation in simple and clear ways. You do not have to worry about pounding your head against complex topics such as 3D matrix transformations in OpenGL; instead, you will learn how to create smooth animations, add PorterDuff compositing into your custom views, and efficiently work with touch events. The little math involved will be broken down, making it

simple. In addition, illustrations will make even the most complex examples clear, and every example will be practical.

How to Use This Book

This book starts with a very broad overview before going into more specific and more advanced topics. As such, it is intended to be read in order, but it is also organized to make reference as easy as possible. Even if you're an advanced developer, it is a good idea to read through all the chapters because of the wide range of material covered; however, you can also jump directly to the topics that most interest you. For example, if you really want to focus on creating your own custom views, you can jump right to Chapter 12, "Developing Custom Views."

This Book's Website

You can find the source code for the examples used throughout this book at https://github.com/IanGClifton/auid2 and the publisher's website at http://www.informit.com/store/android-user-interface-design-implementing-material-9780134191409. From there, you can clone the entire repository, download a full ZIP file, and browse through individual files.

Conventions Used in This Book

This book uses typical conventions found in most programming-related books. Code terms such as class names or keywords appear in `monospace font`. When a class is being referred to specifically (e.g., "Your class should extend the `View` class"), then it will be in monospace font. If it's used more generally (e.g., "When developing a view, don't forget to test on a real device"), then it will not be in a special font.

Occasionally when a line of code is too long to fit on a printed line in the book, a code-continuation arrow (➥) is used to mark the continuation.

You will also see some asides from time to time that present useful information that does not fit into flow of the main text.

> **note**
>
> Notes look like this and are short asides intended to supplement the material in the book with other information you may find useful.

tip

Tips look like this and give you advice on specific topics.

warning

POTENTIAL DATA LOSS OR SECURITY ISSUES Warnings look like this and are meant to bring to your attention to potential issues you may run into or things you should look out for.

This page intentionally left blank

CHAPTER 10

USING ADVANCED TECHNIQUES

The second part of the book covered the full app development process starting with simple ideas and wireframes, turning those into designs, implementing the designs, and tying everything together with beautiful animations. Throughout the process, you relied on users to provide continuous feedback. With all the core work done, most developers consider the process done, but there are a lot of advanced techniques to learn and implement including analyzing and improving the efficiency of your layouts.

Identifying Jank

Unfortunately, development is never as easy as write it once and everything is perfect. In addition to typical bugs that cause the app to crash or misbehave, you're also going to have performance issues to figure out. When you scroll a view and it seems to stutter or hiccup, dropping frames, the experience is bad. These hiccups are sometimes called "jank," which is the opposite of smoothness. You want your app to be as fluid as possible, so eliminating jank can significantly improve the feel of your app.

In many cases, you will see jank but not know what is causing it. For instance, the implementation of the custom view which blurs a portion of an image in the woodworking tools app might be smooth on one device and not smooth on another, causing jank each time a new set of tool images comes into view. What's going on here? If you have difficulty seeing jank, an easy way to help visualize it is by going into the developer options of your phone and turning on Profile GPU rendering (the past few versions of Android have the "On screen as bars" choice so that you don't have to grab the output from `adb`). Go back to the app and scroll around for a bit. A graph of the rendering time will be displayed on top of the UI. The X-axis represents rendering over time and the Y-axis represents the amount of time taken for a frame. The horizontal green line is the limit (16 milliseconds); anything above that line means something is taking too long and causing jank. Figure 10.1 shows an example.

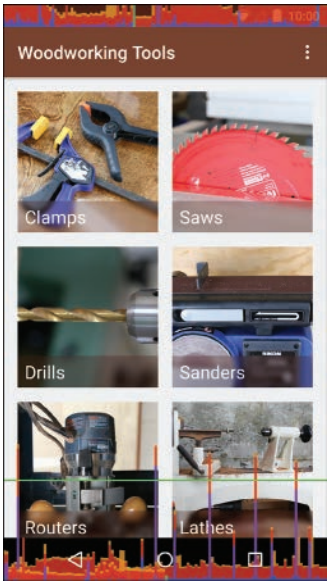

Figure 10.1 Profiling GPU rendering in real time

You should notice that there are multiple graphs on the screen; one for each "window." One represents the status bar, one represents the navigation bar, and one represents the visible

application. Some devices will show the navigation bar and application graph on top of each other, like in Figure 10.1, but you can discern between them by interacting with the app and watching the graph draw itself. The bars are also made of three colors. The purple at the base is used to indicate the draw time; this is the process of converting your draw commands into what's called a display list. A display list is a group of OpenGL commands that have been compiled for execution. Once that's done, the renderer has to execute the display list by communicating with the GPU, represented by the red. At the top of each line is an orange cap. Typically, that cap is small because it represents the amount of time the CPU is waiting for the GPU to acknowledge the commands, which is nearly always fast if you're not doing any custom GPU work.

Using Systrace to Understand Jank

This is a good way to see when the problems happen and you might have some sense of what is going on as you scroll and see the spikes, but it doesn't help much with narrowing down exactly what's going on. That's where the Systrace tool comes in. Systrace logs a significant amount of data about what's going on while it is running and outputs the data as an interactive HTML file. Typically you run it for a brief time (say, 5 seconds), interacting with the app during that time to reproduce the jank, and then you analyze the created HTML file to understand what was taking so long that frames were dropped. Note that not all devices have Systrace available, so it's best used on Nexus devices.

You may recall that in Chapter 8, "Applying the Design," we called out the jank created during scrolling on the main screen of the app. Now it's time to come back to that issue, track down what exactly was causing it with Systrace, and fix it. First, we know that the jank seemed most noticeable when a new item was coming onto the screen. To make this easier to test, we're going to increase the `verticalSpacing` attribute in the `GridView` within `fragment_tool_grid.xml` from 16dp to 200dp. This will add enough spacing between items that only about two rows show up on a typical phone, which means two rows are off screen and can be scrolled on to create jank. One of the most important parts of fixing any bug, whether it's a crash or just a UI hiccup, is being able to easily reproduce it so that you can verify your changes fix it.

First, we can run Systrace without making any other changes. Open Android Device Monitor (under the Tools menu and the Android submenu). You should see your device in the Devices tab on the left. Select it and then pick the confusing Systrace icon above (called out in Figure 10.2). The next window, shown in Figure 10.3, allows you to configure Systrace. Be sure to pick a reasonable name and location. The default 5 seconds and buffer size are both typically fine. You must enable application traces from the app by clicking the dropdown and pick the package name of the app if you want to use any additional logging within your app (it's a good habit to select your app every time). If you don't see the app, be sure that you've selected the correct device and you've installed a debug version of the app (the default build type in Android Studio is a debug build).

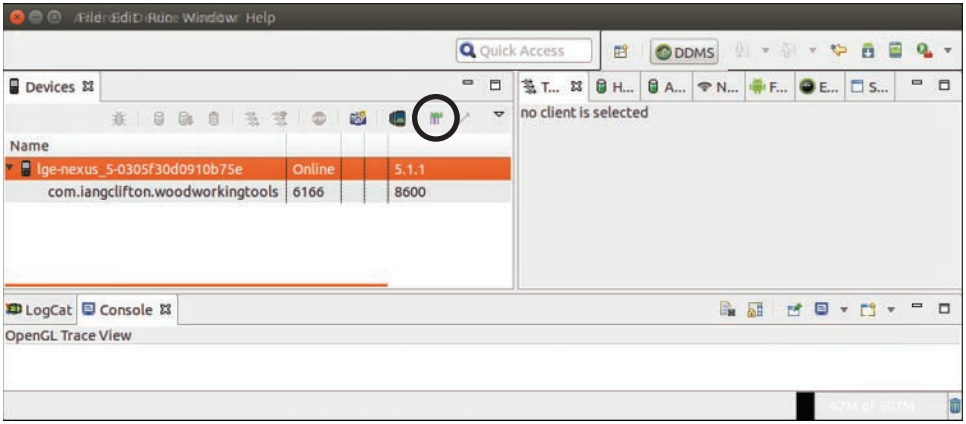

Figure 10.2 The Systrace icon in Android Device Monitor

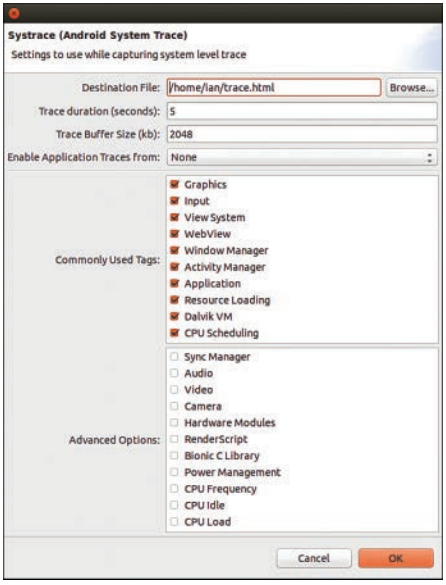

Figure 10.3 The Systrace configuration dialog

Systrace works by tracking the starting and stopping points of events. The events are identified by strings called tags. Android has several built in that are enabled by default as well as some advanced ones that are typically off. For now, leave the commonly used tags all enabled and turn on RenderScript in the advanced options below (remember, we used RenderScript to blur the portion of the image behind the text).

When you click OK, you'll see a dialog informing you that it is collecting trace information. Sometimes it can help to wait briefly (1–2 seconds) before interacting and then start interacting

with the app to reproduce the jank. The wait lets you create a baseline for how long events take when there is no interaction, but it's up to you whether that's useful. Once the time is up, the data is pulled off the device and the HTML file is generated (Systrace currently doesn't tell you that everything is ready; the dialog just goes away and the file appears where you specified). Open it in your browser and prepare to be confused. You should see something like Figure 10.4 in your browser.

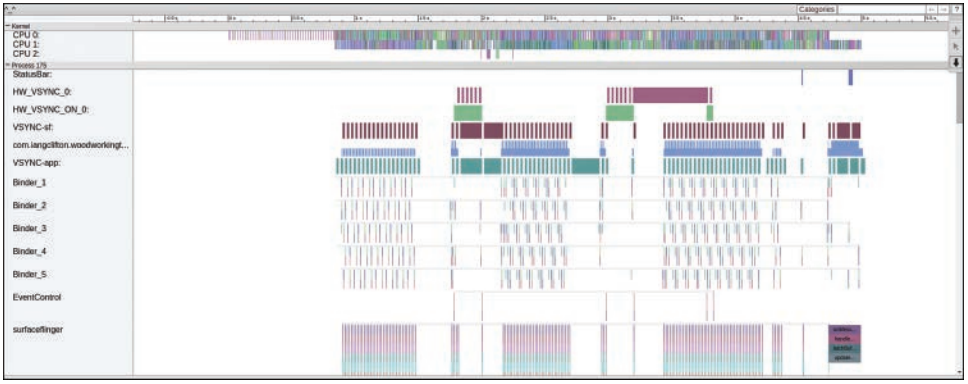

Figure 10.4 The Systrace output in a browser

The HTML will display as a bunch of colorful lines and boxes that make no sense at first. The display shows time along the X-axis and various events along the Y-axis. The wider a box is, the longer it took to perform that operation. At the top will be the CPU info, if you had that enabled. Then you'll have various info, much of which won't be useful to you right now. Further down, you will see "surfaceflinger." SurfaceFlinger takes buffers of pixel data, composites them, and then pushes them to the display. Big gaps in the SurfaceFlinger section when the UI should have been changing (such as while scrolling) are generally problems, but we can scroll down more to the app (the name should be on the left, though it is often cut off for longer names) for more detail.

The controls for the Systrace HTML page are not particularly user friendly. Pretend your Systrace output is as exciting as a first-person shooter and put your fingers on the W, A, S, and D keys. The W and S keys zoom in and out, respectively, while the A and D keys pan. You can also click and drag the mouse up or down to zoom. Double-clicking the mouse creates a guideline under the mouse that you can then position by clicking where you want it. Once you've placed more two guidelines or more, you can see the time between them at the top. At time of writing, the documentation for Systrace is outdated, so the best source for learning the bizarre controls is by clicking the question mark at the very top right when viewing a Systrace file.

Scroll down to the woodworking tools app (it should be the largest section) as shown in Figure 10.5. If you see several huge sections such as `obtainView` that report they never finished, then Systrace is lying to you. The way the tool works is very simple. Each time a developer wants to

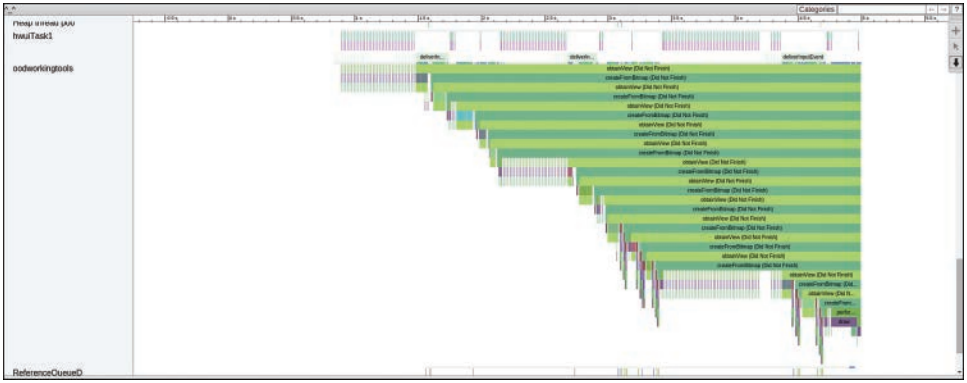

Figure 10.5 The Systrace output showing the app events

track something (including the events that are tracked inside of Android itself), the developer uses the `Trace` class (made available to application developers in Android 4.3; you can also use `TraceCompat` from the support library), calling `beginSection`. When you begin a new section, you give it a name, which shows up in the colored bars here. When that event you were tracking is done, you call `endSection`. Simple, right? The problem is that ending a section always ends the most recently started section. This means that if you forget to call `endSection` such as when you bail out of a method early due to an exception, your section will not end at the right time, if at all. There are a few places where this can happen in the Android system code, unfortunately, so you can receive Systrace output that tells you something like `obtainView` did not finish when you know it did (the view appeared on the screen).

Fortunately, you can "fix" the output of Systrace yourself by calling `endSection` where it was missing. If you see this bug, then after each call to the static `createFromBitmap` method of `Allocation`, call `TraceCompat`'s `endSection` method. This fixes the missing calls in `Allocation` (a bug fix has been accepted for this in the Android source code itself, but it hasn't yet made it into a major release) and now you can run Systrace again. Scrolling down to the woodworking tools app, you should now see only a few rows with most of the entries significantly smaller than they were. The areas where each tracked section is just a sliver are generally good (remember, the X-axis is time) and the larger ones are generally bad. If you use W on your keyboard (or click and drag up), you can zoom in on one of those bigger sections. Double-clicking with the mouse will begin the placement of a timing guideline that you can then position by moving the mouse and place by clicking. Creating two of these will tell you the time between them. Figure 10.6 shows that one of the `obtainView` sections took 155 milliseconds, which is more than eight frames! Keep in mind the power of the device you're testing on as well. In this case, the test was on a Nexus 5. A Nexus 6 might not drop as many frames (a quick test shows it closer to 90 milliseconds), but the Nexus 5 is still in the middle range of Android devices (especially when you look at markets across the world).

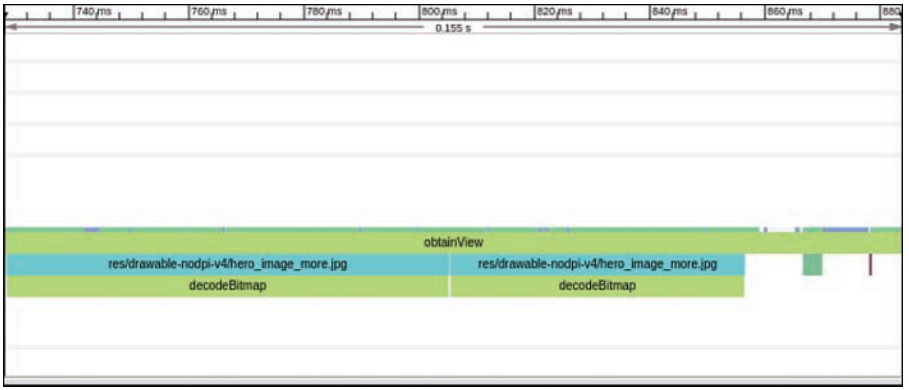

Figure 10.6 Analyzing the details of the Systrace output

The `obtainView` section we've been looking at comes from `AbsListView`'s package-private method of the same name. Because we can be reasonably sure that the Android system code around this is fast, the problem is within the `getView` method of our `ToolGridAdapter`. In the Systrace output, we can also see that the image for the view is being decoded and that alone is taking almost four frames; looking closer, you can see the image is being decoded twice! Just fixing this issue will cut the total `obtainView` time almost in half.

Within the `getView` method, we call the `setImageResource` method of our `CaptionedImageView`. That method simply sets the reference to our drawable and sets the drawable for the image view and then moves on to the blur code. But look a little closer. The drawable reference is being created by inflating the drawable via our resources. The `ImageView` is then having its image set via the resource ID, which causes it to be inflated again. By simply changing that line to instead call `setImageDrawable` and pass in `mDrawable`, we eliminate that second instance of `decodeBitmap`, which was taking around 60 milliseconds.

Maybe you caught this back in Chapter 8, "Applying the Design," but this is the type of mistake that's very easy to make and yet might not be caught. Having a code review process helps increase the chance that this will be caught, but there are still times when it will slip by or you might be working on your own without anyone to review your code. By making use of Android's performance tracking tools, you can catch issues like this, and you will eventually start to get used to keeping your eyes out for these types of problems.

You can run the app with the change and immediately feel the difference, but it's still not smooth, so let's make better use of Systrace. We are reasonably sure that the slowdown is in this area, so let's update the `setImageResource` method so that the first line beings a new trace section and that section is ended in the last line of the method. You can call the sections anything you want, but you can consider a prefix if you're adding a lot of related sections, so

something like "BLUR — setImageResource" tells you that this section is part of the overall blurring process and is specifically the `setImageResource` call. Diving into the `updateBlur` method, there are a lot of different places that might be the cause of slowdown. This is a case where intuition and experience will help you decide, but you can't go wrong with extra logging.

Anywhere that you create or manipulate a bitmap is a good place to track. For instance, the two lines that create the `portionToBlur` and `blurredBitmap` objects can be wrapped with `TraceCompat` calls (that is, put a call to `beginSection` before those objects are created and a call to `endSection` after). Wrapping the full RenderScript section is also a good idea. You can even add some sections within the RenderScript portion of the code to better understand what is taking time there. Listing 10.1 shows an example of the `setImageResource` and `updateBlur` methods.

Listing 10.1 An Example of Methods with Systrace Logging Added

```
public void setImageResource(@DrawableRes int drawableResourceId) {
    TraceCompat.beginSection("BLUR - setImageResource");
    mDrawable = getResources().getDrawable(drawableResourceId);
    mImageView.setImageDrawable(mDrawable);
    updateBlur();
    TraceCompat.endSection();
}

private void updateBlur() {
    if (!(mDrawable instanceof BitmapDrawable)) {
        return;
    }
    final int textViewHeight = mTextView.getHeight();
    if (textViewHeight == 0) {
        return;
    }

    // Determine the size of the TextView compared to the height of the
➥ ImageView
    final float ratio = (float) textViewHeight / mImageView.getHeight();

    // Get the Bitmap
    final BitmapDrawable bitmapDrawable = (BitmapDrawable) mDrawable;
    final Bitmap originalBitmap = bitmapDrawable.getBitmap();

    // Calculate the height as a ratio of the Bitmap
    int height = (int) (ratio * originalBitmap.getHeight());

    // The y position is the number of pixels height represents from
➥ the bottom of the Bitmap
    final int y = originalBitmap.getHeight() - height;
```

```
    TraceCompat.beginSection("BLUR - createBitmaps");
    final Bitmap portionToBlur = Bitmap.createBitmap(originalBitmap, 0,
➥ y, originalBitmap.getWidth(), height);
    final Bitmap blurredBitmap = portionToBlur.copy(Bitmap.
➥ Config.ARGB_8888, true);
    TraceCompat.endSection();

    // Use RenderScript to blur the pixels
    TraceCompat.beginSection("BLUR - RenderScript");
    RenderScript rs = RenderScript.create(getContext());
    ScriptIntrinsicBlur theIntrinsic = ScriptIntrinsicBlur.create(rs,
➥ Element.U8_4(rs));
    TraceCompat.beginSection("BLUR - RenderScript Allocation");
    Allocation tmpIn = Allocation.createFromBitmap(rs, portionToBlur);
    // Fix internal trace that isn't ended
    TraceCompat.endSection();
    Allocation tmpOut = Allocation.createFromBitmap(rs,
➥ blurredBitmap);
    // Fix internal trace that isn't ended
    TraceCompat.endSection();
    TraceCompat.endSection();
    theIntrinsic.setRadius(25f);
    theIntrinsic.setInput(tmpIn);
    TraceCompat.beginSection("BLUR - RenderScript forEach");
    theIntrinsic.forEach(tmpOut);
    TraceCompat.endSection();
    TraceCompat.beginSection("BLUR - RenderScript copyTo");
    tmpOut.copyTo(blurredBitmap);
    TraceCompat.endSection();
    new Canvas(blurredBitmap).drawColor(mScrimColor);
    TraceCompat.endSection();

    // Create the new bitmap using the old plus the blurred portion and
➥ display it
    TraceCompat.beginSection("BLUR - Finalize image");
    final Bitmap newBitmap = originalBitmap.copy(Bitmap.
➥ Config.ARGB_8888, true);
    final Canvas canvas = new Canvas(newBitmap);
    canvas.drawBitmap(blurredBitmap, 0, y, new Paint());
    mImageView.setImageBitmap(newBitmap);
    TraceCompat.endSection();
}
```

Running the app and using Systrace again, we can see that a particular instance of the `obtainView` call is now taking 90 milliseconds with the `decodeBitmap` portion roughly two-thirds of that. The RenderScript portion of our code is the majority of the rest of the time with almost half of the RenderScript time coming from the `copyTo` call. Let's start with the image loading portion, because that's the larger part.

Optimizing Images

There are a few big things you can do to speed up your use of images. First, you can shrink the image file sizes by stripping out metadata and adjusting the compression. Second, you can size them correctly (loading an image that's bigger than it is displayed is obviously slower than loading one that's exactly the size that will be displayed). Finally, you can keep them in memory to avoid loading them again.

Shrinking Images

Many people aren't aware that images can contain a lot more data than they need. For instance, JPEGs can be "progressive," which means they actually store multiple versions of the image at progressively higher levels of detail. This was particularly helpful for old webpages on dialup, when you wanted some sense of what the image was before all of it could be downloaded on the slow connection, but it causes the images to be much larger and Android doesn't use this extra data because the full image is already on the local disk. JPEGs can also store metadata in EXIF or XMP formats. These are two different formats for storing information about the photo such as when it was taken, what camera was used, and even how long the photo was exposed. That data is great for photographers but it adds bulk to the image that isn't helpful for our app, so it can be stripped out. Another common inclusion in JPEGs is actually a thumbnail of the image. This is intended to help file browsers and cameras themselves to display a thumbnail without having to read in massive images and scale them; it can be stripped out as well.

You can also adjust the compression of the image. Although a more compressed image will still take up the same amount of memory once it is decompressed and decoded, it can significantly decrease the loading time because the amount of data read from disk can be substantially decreased. JPEG is a lossy format, which means you lose image detail by compressing it, but you can decrease file sizes substantially with only minor changes in compression. Typically, the compression level is described in terms of quality where 100 is highest quality and least compression. Although you might want to keep quality closer to 100 for your own personal photos, that isn't necessary for most app images like what are in the woodworking tools app. The amount of compression you can use while maintaining quality will vary depending on what is in the image (e.g., details like thin letters will look poor faster than something like a tree in the background), but you can generally start at a quality level of 90 and see how it looks.

By stripping out all this extra data and changing the quality level to 90, we decrease the hero image sizes in the woodworking tools app substantially. For instance, the image of the clamps was 493KB and it is now 272KB. The smallest image (in terms of file size) was the drill at 165KB and even it shrank down to 64KB. Testing the app again shows that the time to decode the images has gone from about 60 milliseconds to about half that (with a fair bit of variation). Photoshop makes this pretty easy by picking "Save for Web" and ensuring that the quality is set how you want, it is not progressive, and no metadata is enabled. The process in GIMP is slightly different, starting with the "Export As" command that will give you an "Export Image as JPEG"

dialog when saving as a JPEG; the dialog has "Advanced Options" where you want to make sure the image is not progressive, does not contain EXIF or XMP data, and does not have an embedded thumbnail.

Although these images are JPEGs, the same idea of shrinking down the file size applies to PNGs. PNGs are lossless, so you should always use the maximum compression. PNGs can be 8-bit (256 colors), 24-bit (red, green, and blue channels of 256 values each), or 32-bit (adding the alpha channel with 256 values). You can also save them with custom palettes, which can significantly shrink down the overall size. Many graphics programs support shrinking PNGs quite a bit themselves, so look for those options (such as a "save for web" feature). There are also third-party tools like "pngcrush," which will try many different ways of compressing and shrinking the image to give you a smaller file size.

Using the Right Sizes

Our "hero" images are being used as thumbnails, so they're actually significantly larger than what is displayed. The screen has a 16sp space on the left, center, and right of the thumbnail grid. In the case of the Nexus 5 (an XXHDPI device, which means the spaces are 48px), the spaces use a total of 144px. With a screen width of 1080px, we can subtract the 144px of space and divide the remainder by 2 (because there are two thumbnails in each row) to determine the thumbnails will be 468px in each dimension. That means we're loading a 1080 × 607 image (roughly 650,000px) when we really just need a 468 × 468 image (about 220,000px). If we run the same numbers for a Nexus 4, which is an XHDPI device with a screen width of 768px, we need an image that's 336px (about 110,000px). That means a Nexus 5 is loading almost three times as many pixels as needed and a Nexus 4 is loading almost six times as many pixels as needed! Not only is using larger images than necessary slower to load, it's slower to blur and takes up more memory.

There is a simple method we can use to handle this. First, we will create a few sizes to handle the most typical device configurations. Then we'll store all of these in the `drawables-nodpi` folder with a filename that indicates the size (for instance, `hero_image_clamps_468.png`) because we don't want Android to scale these images. It's a good idea to make sure the original images have updated names (e.g., `hero_image_clamps_1080.png`) to be consistent, and don't forget to make sure the images you save out have any extra data stripped (they should not be progressive JPEGs, they shouldn't have EXIF data, XMP data, or thumbnails embedded). Removing the extra data helps keep your APK smaller and also slightly decreases the loading time for the images. After that we just need some simple code to pick the best size.

Listing 10.2 demonstrates a simple `BitmapUtils` class that picks the best sized image. You might be wondering if there's a way to construct the drawable resource ID from a string. For instance, you could store the string "hero_image_clamps_" and then concatenate the appropriate size, then get the resource that references. Indeed, that is possible. With a reference to your resources like you get from `getResources()`, you can use the `getIdentifier` method, but that method actually uses reflection to figure out which resource you're

referencing, so it is slow (defeating the purpose of optimizing this code in the first place). You should typically avoid using that method.

Listing 10.2 A `BitmapUtils` Class

```
public class BitmapUtils {

    private static final int THUMBNAIL_SIZE_336 = 336;
    private static final int THUMBNAIL_SIZE_468 = 468;

    public static int getScreenWidth(@NonNull Context context) {
        WindowManager windowManager = (WindowManager) context.
➥ getSystemService(Context.WINDOW_SERVICE);
        Display display = windowManager.getDefaultDisplay();
        Point point = new Point();
        display.getSize(point);
        return point.x;
    }

    /**
     * Returns a resource ID to a smaller version of the drawable, when
➥ possible.
     *
     * This is intended just for the hero images. If a smaller size of
➥ the resource ID cannot
     * be found, the original resource ID is returned.
     *
     * @param resourceId int drawable resource ID to look for
     * @param desiredSize int desired size in pixels of the drawable
     * @return int drawable resource ID to use
     */
    @DrawableRes
    public static int getPresizedImage(@DrawableRes int resourceId, int
➥ desiredSize) {
        switch (resourceId) {
            case R.drawable.hero_image_clamps_1080:
                if (desiredSize <= THUMBNAIL_SIZE_336) {
                    return R.drawable.hero_image_clamps_336;
                } else if (desiredSize <= THUMBNAIL_SIZE_468) {
                    return R.drawable.hero_image_clamps_468;
                }
                break;
            case R.drawable.hero_image_saw_1080:
                if (desiredSize <= THUMBNAIL_SIZE_336) {
                    return R.drawable.hero_image_saw_336;
                } else if (desiredSize <= THUMBNAIL_SIZE_468) {
                    return R.drawable.hero_image_saw_468;
                }
                break;
```

```
            case R.drawable.hero_image_drill_1080:
                if (desiredSize <= THUMBNAIL_SIZE_336) {
                    return R.drawable.hero_image_drill_336;
                } else if (desiredSize <= THUMBNAIL_SIZE_468) {
                    return R.drawable.hero_image_drill_468;
                }
                break;
            case R.drawable.hero_image_sander_1080:
                if (desiredSize <= THUMBNAIL_SIZE_336) {
                    return R.drawable.hero_image_sander_336;
                } else if (desiredSize <= THUMBNAIL_SIZE_468) {
                    return R.drawable.hero_image_sander_468;
                }
                break;
            case R.drawable.hero_image_router_1080:
                if (desiredSize <= THUMBNAIL_SIZE_336) {
                    return R.drawable.hero_image_router_336;
                } else if (desiredSize <= THUMBNAIL_SIZE_468) {
                    return R.drawable.hero_image_router_468;
                }
                break;
            case R.drawable.hero_image_lathe_1080:
                if (desiredSize <= THUMBNAIL_SIZE_336) {
                    return R.drawable.hero_image_lathe_336;
                } else if (desiredSize <= THUMBNAIL_SIZE_468) {
                    return R.drawable.hero_image_lathe_468;
                }
                break;
            case R.drawable.hero_image_more_1080:
                if (desiredSize <= THUMBNAIL_SIZE_336) {
                    return R.drawable.hero_image_more_336;
                } else if (desiredSize <= THUMBNAIL_SIZE_468) {
                    return R.drawable.hero_image_more_468;
                }
                break;
        }

        return resourceId;
    }
}
```

Testing out loading images that are the correct size for a Nexus 5 shows that we drop from the 30 milliseconds we saw after shrinking the file sizes in the previous section to just 7 milliseconds! This has also made our blurring code significantly faster (down from 30 milliseconds to about 15 milliseconds) because it's operating on fewer pixels. Overall, it is still taking about 22 milliseconds to get the view, so it's still slow, but the improvements so far have been substantial.

Before we move on, it's worth noting that you can also tackle this by loading the images at a smaller size using the `BitmapFactory` class, which supports subsampling. By subsampling, you can read a fraction of the number of pixels. For instance, if you have an image that was 1000px wide but you only need it at 500px wide, you can read every other pixel. This isn't as efficient as providing correctly sized images, but it is more universal.

The general idea is that you read in just the metadata about the image (which includes the dimensions) by making use of the `Options` class. Create a new instance of the `Options` class and set `inJustDecodeBounds` to true. Now when you decode the image with `decodeResources`, you use the `Options` object to tell it that you only want the bounds (the size of the image). The `Options` object will now have its `outWidth`, `outHeight`, and `outMimeType` set to the values of the actual image (and `BitmapFactory` will return null).

Once you know how big the image is, you figure out how to subsample the image. The `BitmapFactory` class works in powers of 2, so you can subsample every second pixel, every fourth pixel, every eighth pixel, and so forth. Using a simple while loop, we can double the sample size until the resulting image will be just larger than the desired width. Listing 10.3 shows a simple implementation of this method.

Listing 10.3 The `getSizedBitmap` Method

```
public static Bitmap getSizedBitmap(@NonNull Resources res,
➥ @DrawableRes int resId, int desiredWidth) {
    // Load just the size of the image
    BitmapFactory.Options options = new BitmapFactory.Options();
    options.inJustDecodeBounds = true;
    BitmapFactory.decodeResource(res, resId, options);

    // Options now has the bounds; prepare it for getting the actual
➥ image
    options.inSampleSize = 1;
    options.inJustDecodeBounds = false;

    if (options.outWidth > desiredWidth) {
        final int halfWidth = options.outWidth / 2;

        while (halfWidth / options.inSampleSize > desiredWidth) {
            options.inSampleSize *= 2;
        }
    }

    return BitmapFactory.decodeResource(res, resId, options);
}
```

This is a simple way of making your image loading more efficient and this technique can be used throughout your app; however, it isn't as efficient as having your images exactly the correct size.

Testing image loading with this method, the average speed is around 15 milliseconds (with a fair bit of variation), compared to the 30 milliseconds without this method or the 7 milliseconds with properly sized images.

Using an Image Cache

Every time you load an image from disk, it has to be decoded again. As we've seen, this process is often very slow. Worse, we can be loading the same image multiple times because the view that was displaying it goes off the screen and garbage collection is triggered and later it comes back on screen. An image cache allows you to specify a certain amount of memory to use for images, and the typical pattern is an LRU (least-recently-used) cache. As you put images in and ask for images out, the cache keeps track of that usage. When you put in a new image and there isn't enough room for it, the cache will evict the images that haven't been used for the longest amount of time, so that images you just used (and are most likely to still be relevant) stay in memory. When you load images, they go through the image cache, which will keep them in memory as long as the cache has room. That means the view going off screen and coming back on might not cause the image to have to be read from disk, which can save several milliseconds.

You can use the `LruCache` class that is in the support library to work with any version of Android. It has two generics, one for the keys and one for the values (the objects you cache). For images, you'll typically have a string key (such as a URL and filename), but you can use whatever makes sense for your needs. Because the `LruCache` class is designed to work in many situations, the concept of "size" can be defined by you. By default, the size of the cache is determined by the number of objects, but that doesn't make sense for our use. A single large image will take up a lot more memory than several smaller ones, so we can make the size measured in kilobytes. We need to override the `sizeOf` method to return the number of kilobytes a given entry is. We also need to decide on the maximum size of the cache. This will depend heavily on the type of app you're developing, but we can make this a portion of the overall memory available to our app. Let's start with an eighth of the total memory. We can also place an upper limit on the size of 16mb so that our app is well behaved on devices that give each VM much more memory that we necessarily need. Listing 10.4 shows this simple class.

Listing 10.4 The `BitmapCache` Class

```
public class BitmapCache extends LruCache<String, Bitmap> {
    public static final String TAG = "BitmapCache";

    private static final int MAXIMUM_SIZE_IN_KB = 1024 * 16;

    public BitmapCache() {
        super(getCacheSize());
    }

    @Override
    protected int sizeOf(String key, Bitmap bitmap) {
        return bitmap.getByteCount() / 1024;
    }
```

```
    /**
     * Returns the size of the cache in kilobytes
     *
     * @return int total kilobytes to make the cache
     */
    private static int getCacheSize() {
        // Maximum KB available to the VM
        final int maxMemory = (int) (Runtime.getRuntime().maxMemory()
➥ / 1024);
        // The smaller of an eighth of the total memory or 16MB
        final int cacheSize = Math.min(maxMemory / 8, MAXIMUM_SIZE_IN_KB);
        Log.v(TAG, "BitmapCache size: " + cacheSize + "kb");
        return cacheSize;
    }
}
```

We can create the `BitmapCache` in our `BitmapUtils` class and then add some simple methods for interacting with it. In the case of our grid of images, we actually care more about the versions of the images that have been blurred along the bottom already, so we can directly cache those. Listing 10.5 shows the `BitmapUtils` methods we've added and Listing 10.6 shows the updated methods in `CaptionedImageView`.

Listing 10.5 The `BitmapUtils` Class Updated to Use the Cache

```
publicclass BitmapUtils {

    private static final int THUMBNAIL_SIZE_336 = 336;
    private static final int THUMBNAIL_SIZE_468 = 468;

    private static final BitmapCache BITMAP_CACHE = new BitmapCache();

    public synchronized static void cacheBitmap(@NonNull String key,
➥ @NonNull Bitmap bitmap) {
        BITMAP_CACHE.put(key, bitmap);
    }

    public synchronized static Bitmap getBitmap(@NonNull String key) {
        return BITMAP_CACHE.get(key);
    }

    public synchronized static Bitmap getBitmap(@NonNull Resources res,
➥ @DrawableRes int resId) {
        String key = String.valueOf(resId);
        Bitmap bitmap = BITMAP_CACHE.get(key);
        if (bitmap == null) {
            bitmap = BitmapFactory.decodeResource(res, resId);
            BITMAP_CACHE.put(key, bitmap);
```

```
        }
        return bitmap;
    }

    public static int getScreenWidth(@NonNull Context context) {
        WindowManager windowManager = (WindowManager) context.
➥getSystemService(Context.WINDOW_SERVICE);
        Display display = windowManager.getDefaultDisplay();
        Point point = new Point();
        display.getSize(point);
        return point.x;
    }

    /**
     * Returns a resource ID to a smaller version of the drawable, when
➥possible.
     *
     * This is intended just for the hero images. If a smaller size of
➥the resource ID cannot
     * be found, the original resource ID is returned.
     *
     * @param resourceId int drawable resource ID to look for
     * @param desiredSize int desired size in pixels of the drawable
     * @return int drawable resource ID to use
     */
    @DrawableRes
    public static int getPresizedImage(@DrawableRes int resourceId, int
➥desiredSize) {
        switch (resourceId) {
            case R.drawable.hero_image_clamps_1080:
                if (desiredSize <= THUMBNAIL_SIZE_336) {
                    return R.drawable.hero_image_clamps_336;
                } else if (desiredSize <= THUMBNAIL_SIZE_468) {
                    return R.drawable.hero_image_clamps_468;
                }
                break;
            case R.drawable.hero_image_saw_1080:
                if (desiredSize <= THUMBNAIL_SIZE_336) {
                    return R.drawable.hero_image_saw_336;
                } else if (desiredSize <= THUMBNAIL_SIZE_468) {
                    return R.drawable.hero_image_saw_468;
                }
                break;
            case R.drawable.hero_image_drill_1080:
                if (desiredSize <= THUMBNAIL_SIZE_336) {
                    return R.drawable.hero_image_drill_336;
                } else if (desiredSize <= THUMBNAIL_SIZE_468) {
                    return R.drawable.hero_image_drill_468;
                }
```

```
                break;
            case R.drawable.hero_image_sander_1080:
                if (desiredSize <= THUMBNAIL_SIZE_336) {
                    return R.drawable.hero_image_sander_336;
                } else if (desiredSize <= THUMBNAIL_SIZE_468) {
                    return R.drawable.hero_image_sander_468;
                }
                break;
            case R.drawable.hero_image_router_1080:
                if (desiredSize <= THUMBNAIL_SIZE_336) {
                    return R.drawable.hero_image_router_336;
                } else if (desiredSize <= THUMBNAIL_SIZE_468) {
                    return R.drawable.hero_image_router_468;
                }
                break;
            case R.drawable.hero_image_lathe_1080:
                if (desiredSize <= THUMBNAIL_SIZE_336) {
                    return R.drawable.hero_image_lathe_336;
                } else if (desiredSize <= THUMBNAIL_SIZE_468) {
                    return R.drawable.hero_image_lathe_468;
                }
                break;
            case R.drawable.hero_image_more_1080:
                if (desiredSize <= THUMBNAIL_SIZE_336) {
                    return R.drawable.hero_image_more_336;
                } else if (desiredSize <= THUMBNAIL_SIZE_468) {
                    return R.drawable.hero_image_more_468;
                }
                break;
        }

        return resourceId;
    }

    public static Bitmap getSizedBitmap(@NonNull Resources res,
➥ @DrawableRes int resId, int desiredWidth) {
        // Load just the size of the image
        BitmapFactory.Options options = new BitmapFactory.Options();
        options.inJustDecodeBounds = true;
        BitmapFactory.decodeResource(res, resId, options);

        // Options now has the bounds; prepare it for getting the
➥ actual image
        options.inSampleSize = 1;
        options.inJustDecodeBounds = false;

        if (options.outWidth > desiredWidth) {
            final int halfWidth = options.outWidth / 2;

            while (halfWidth / options.inSampleSize > desiredWidth) {
```

```
                options.inSampleSize *= 2;
            }
        }

        return BitmapFactory.decodeResource(res, resId, options);
    }
}
```

Listing 10.6 The Updated `CaptionedImageView` Methods

```
public void setImageResource(@DrawableRes int drawableResourceId) {
    TraceCompat.beginSection("BLUR - setImageResource");
    mDrawableResourceId = drawableResourceId;
    Bitmap bitmap = BitmapUtils.getBitmap(getResources(),
➥ mDrawableResourceId);
    mDrawable = new BitmapDrawable(getResources(), bitmap);
    mImageView.setImageDrawable(mDrawable);
    updateBlur();
    TraceCompat.endSection();
}

private void updateBlur() {
    if (!(mDrawable instanceof BitmapDrawable)) {
        return;
    }
    final int textViewHeight = mTextView.getHeight();
    final int imageViewHeight = mImageView.getHeight();
    if (textViewHeight == 0 || imageViewHeight == 0) {
        return;
    }

    // Get the Bitmap
    final BitmapDrawable bitmapDrawable = (BitmapDrawable) mDrawable;
    final Bitmap originalBitmap = bitmapDrawable.getBitmap();

    // Determine the size of the TextView compared to the height of the
➥ ImageView
    final float ratio = (float) textViewHeight / imageViewHeight;

    // Calculate the height as a ratio of the Bitmap
    final int height = (int) (ratio * originalBitmap.getHeight());
    final int width = originalBitmap.getWidth();
    final String blurKey = getBlurKey(width);
    Bitmap newBitmap = BitmapUtils.getBitmap(blurKey);
    if (newBitmap != null) {
        mImageView.setImageBitmap(newBitmap);
        return;
    }
```

```
    // The y position is the number of pixels height represents from
➥ the bottom of the Bitmap
    final int y = originalBitmap.getHeight() – height;

    TraceCompat.beginSection("BLUR – createBitmaps");
    final Bitmap portionToBlur = Bitmap.createBitmap(originalBitmap,
➥ 0, y, originalBitmap.getWidth(), height);
    final Bitmap blurredBitmap = Bitmap.createBitmap(portionToBlur.
getWidth(), height, Bitmap.Config.ARGB_8888);
    TraceCompat.endSection();

    // Use RenderScript to blur the pixels
    TraceCompat.beginSection("BLUR – RenderScript");
    RenderScript rs = RenderScript.create(getContext());
    ScriptIntrinsicBlur theIntrinsic = ScriptIntrinsicBlur.create(rs,
➥ Element.U8_4(rs));
    TraceCompat.beginSection("BLUR – RenderScript Allocation");
    Allocation tmpIn = Allocation.createFromBitmap(rs, portionToBlur);
    // Fix internal trace that isn't ended
    TraceCompat.endSection();
    Allocation tmpOut = Allocation.createFromBitmap(rs,
➥ blurredBitmap);
    // Fix internal trace that isn't ended
    TraceCompat.endSection();
    TraceCompat.endSection();
    theIntrinsic.setRadius(25f);
    theIntrinsic.setInput(tmpIn);
    TraceCompat.beginSection("BLUR – RenderScript forEach");
    theIntrinsic.forEach(tmpOut);
    TraceCompat.endSection();
    TraceCompat.beginSection("BLUR – RenderScript copyTo");
    tmpOut.copyTo(blurredBitmap);
    TraceCompat.endSection();
    new Canvas(blurredBitmap).drawColor(mScrimColor);
    TraceCompat.endSection();

    // Create the new bitmap using the old plus the blurred portion and
➥ display it
    TraceCompat.beginSection("BLUR – Finalize image");
    newBitmap = originalBitmap.copy(Bitmap.Config.ARGB_8888, true);
    final Canvas canvas = new Canvas(newBitmap);
    canvas.drawBitmap(blurredBitmap, 0, y, new Paint());
    BitmapUtils.cacheBitmap(blurKey, newBitmap);
    mTextView.setBackground(null);
    mImageView.setImageBitmap(newBitmap);
    TraceCompat.endSection();
}
```

Given that on a typical device six of the seven images in the woodworking tools app are available on the first screen (once the spacing is back to normal); you might consider pre-caching the seventh image. By precaching them all, you will slightly increase the loading time of the app, but you will ensure that the scrolling is smooth. If you had even more images to handle, you might even move the image loading and blurring to a background thread. Although RenderScript is extremely fast, it still takes time pass the data to the GPU, process it, and pass that data back, so it's not a bad idea to push that work to a background thread if you're going to be doing it often.

Additional Performance Improvements

There are many causes of performance issues, but they are nearly always related to doing too much work on the UI thread. Sometimes you can affect these issues directly (like moving the loading of images to a background thread) and other times you have to be less direct (like maintaining a reference to an object you don't need to avoid garbage collection during an animation). Whether you have to be direct or not, knowing additional techniques to improve performance is always beneficial.

Controlling Garbage Collection

Garbage collection is one of those things that you can simultaneously love and hate. It's great because it lets you focus on the fun part of developing an app rather than worrying about tedious tasks like reference counting. It's horrible because it slows down your app when it happens. Although Android has had concurrent garbage collection for years now and its introduction helped a huge amount, garbage collection can still cause jank. Remember, you have just 16 milliseconds for each frame, so the garbage collector taking just 4 milliseconds is effectively taking 25% of your time for that frame.

Although you can't do much to control the garbage collector directly other than just triggering a collection, you can adjust the way you code your app to decrease the work it does or even simply change the timing. In particular, you should be conscious of where you are allocating memory and where you are releasing it. Every time you allocate memory, such as declaring an object or array, you are asking the system for a consecutive chunk of free memory. The fact that it's consecutive is important because that means even if you have 10mb free, you could ask for 500kb and still incur the cost of the VM reorganizing the memory if there isn't enough consecutive free memory available. Even if the virtual machine doesn't have to clear out memory, just moving chunks of memory around to make a large, consecutive section takes time. On the other side of that, every time you get rid of a reference to an object or an array and that was the last reference to that object or array, there is a chance the garbage collector will run. Halfway done with that animation and now you don't need those giant bitmaps? Removing your reference could cause the animation to pause or skip frames.

In general, you don't want to allocate or release memory in any method that is already potentially slow or gets called many times in succession. For instance, be careful in the `onDraw` and `onLayout` methods of `View` (both are covered in Chapter 13, "Developing Fully Custom Views"), `getView` of `Adapter` implementations, and any methods you trigger during animations. If you have to allocate objects, consider whether it's possible to keep them around or reuse them (such as in an object pool).

View Holder Pattern

`ListView` and other `AbsListView` implementations are excellent ways to effectively display a subset of a larger data set. They effectively reuse views to limit garbage collection and keep everything smooth, but it's still easy to run into problems. One of the most common methods to use within a `getView` call of an adapter is `findViewById`. For example, look at Listing 10.7 that shows a simple `getView` implementation.

Listing 10.7 An Example `getView` Method

```
public View getView(int position, View convertView, ViewGroup parent) {
    if (convertView == null) {
        convertView = mLayoutInflater.inflate(R.layout.list_item,
➥ parent, false);
    }

    ListItem listItem = getItem(position);
    ImageView imageView = (ImageView) convertView.findViewById
➥ (R.id.imageView);
    Drawable drawable = mContext.getDrawable(R.drawable.person);
    drawable.setTintMode(PorterDuff.Mode.SRC_ATOP);
    drawable.setTint(listItem.getColor());
    imageView.setImageDrawable(drawable);

    TextView textView = (TextView) convertView.findViewById
➥ (R.id.count);

    textView.setText(listItem.getCount());
    textView = (TextView) convertView.findViewById(R.id.title);
    textView.setText(listItem.getTitle());
    textView = (TextView) convertView.findViewById(R.id.subtitle);
    textView.setText(listItem.getSubtitle());

    return convertView;
}
```

Each `getView` call traverses the view to find the views that need to be updated. In this case, there are four separate `findViewById` calls. Each call will first check the `convertView` ID, then the children one at a time until a match is found. If this example has just the

`convertView` plus the four views that are searched for, the first call will check two views to get the match, the next three, and so on. This leads to 14 view lookups in this simple case! The more complex the view, the longer it takes to traverse the hierarchy and find the views you are looking for.

The ideal solution would be to only have to find the views one time and then just retain the references, and that's what the view holder pattern does. Because you are reusing views (via the convert view), you have to traverse a finite number of views before you have found every view you care about in the list. By creating a class called `ViewHolder` that has references to each of the views you care about, you can instantiate that class once per view in the `ListView` and then reuse that class as much as needed. This class is implemented as a static inner class, so it is really just acting as a container for view references. See Listing 10.8 for a simple example of a class that takes the view and sets all the necessary references.

Listing 10.8 An Example of a `ViewHolder` Class

```
private static class ViewHolder {
    /*package*/final ImageView imageView;
    /*package*/final TextView count;
    /*package*/final TextView title;
    /*package*/final TextView subtitle;

    /*package*/ ViewHolder(View v) {
        imageView = (ImageView) v.findViewById(R.id.imageView);
        count = (TextView) v.findViewById(R.id.count);
        title = (TextView) v.findViewById(R.id.title);
        subtitle = (TextView) v.findViewById(R.id.subtitle);
    }
}
```

Looking back at the `getView` method, a few minor changes can significantly improve the efficiency. If `convertView` is null, inflate a new view as we already have and then create a new `ViewHolder` instance, passing in the view we just created. To keep the `ViewHolder` with this view, we call the `setTag` method, which allows us to associate an arbitrary object with any view. If `convertView` is not null, we simply call `getTag` and cast the result to the `ViewHolder`.

Now that we have the `ViewHolder`, we can just reference the views directly, so the rest of the `getView` code is not only much simpler than before but also better performing and easier to read. See Listing 10.9 for the updated `getView` call. Any time you use an adapter and you use the child views in the `getView` method, you should use this view holder pattern. For example, the `ToolArrayAdapter` in the woodworking tools app should be updated to use the view holder pattern; give it a try.

Listing 10.9 The `getView` Method Using a `ViewHolder`

```
public View getView(int position, View convertView, ViewGroup parent) {
    ViewHolder viewHolder;
    if (convertView == null) {
        convertView = mLayoutInflater.inflate(R.layout.list_item,
➥ parent, false);
        viewHolder = new ViewHolder(convertView);
        convertView.setTag(viewHolder);
    } else {
        viewHolder = (ViewHolder) convertView.getTag();
    }

    ListItem listItem = getItem(position);
    Drawable drawable = mContext.getDrawable(R.drawable.person);
    drawable.setTintMode(PorterDuff.Mode.SRC_ATOP);
    drawable.setTint(listItem.getColor());
    viewHolder.imageView.setImageDrawable(drawable);

    viewHolder.count.setText(listItem.getCount());
    viewHolder.title.setText(listItem.getTitle());
    viewHolder.subtitle.setText(listItem.getSubtitle());

    return convertView;
}
```

Eliminating Overdraw

Overdraw is when your app causes pixels to be drawn on top of each other. For example, imagine a typical app with a background, whether plain or an image. Now you put an opaque button on it. First, the device draws the background; then it draws the button. The background under the button was drawn but is never seen, so that processing and data transfer are wasted.

You might wonder how you can actually eliminate overdraw then, and the answer is that you do not need to. You only need to eliminate excessive overdraw. What "excessive" means is different for each device, but the general rule of thumb is that you should not be drawing more than three times the number of pixels on the screen (as detailed in Chapter 7, "Designing the Visuals"). When you go above three times the number of pixels, performance often suffers.

It's worth noting that some devices are better than others at efficiently avoiding drawing pixels when opaque pixels would be drawn right on top of them. GPUs that use deferred rendering are able to eliminate overdraw in cases where fully opaque pixels are drawn on fully opaque pixels, but not all Android devices have GPUs that use deferred rendering. Further, if pixels have any amount of transparency, that overdraw cannot be eliminated because the pixels have to be

combined. That is why designs that contain a significant amount of transparency are inherently more difficult to make smooth and efficient than designs that do not.

Overdraw is easiest to eliminate when you can see it. Android 4.2 and newer offer a developer option to show GPU overdraw by coloring the screen differently based on how many times a pixel has been drawn and redrawn. To enable it, go to the device settings and then Developer options and scroll to the "drawing" section to enable the Show GPU Overdraw option (see Figure 10.7). When this option is checked, apps will be colored to show the amount of overdraw. Current versions of Android don't require restarting the app, but older versions do.

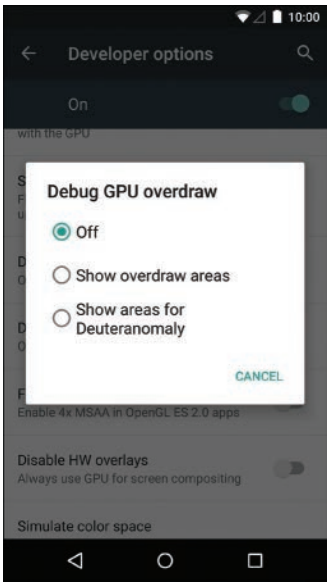

Figure 10.7 The Show GPU Overdraw option in Android's developer options

First, you should understand what the color tints mean. If there is no tint, there is no overdraw, and this is the ideal situation. A blue tint indicates a single overdraw (meaning the pixel was drawn once and then drawn again), and you can think of it as being "cold" because your device can easily handle a single level of overdraw (so the processor is not overheating). When something is tinted green, it has been overdrawn twice. Light red indicates an overdraw of three times, and dark red indicates an overdraw of four (or more) times (red, hot, bad!).

Large sections of blue are acceptable as long as the whole app is not blue (if it were, that'd suggest you're drawing the full screen and immediately redrawing it, which is very wasteful). Medium-sized sections of green are okay, but you should avoid having more than half of the screen green. Light red is much worse, but it's still okay for small areas such as text or a tiny icon. Dark red should make you cry. Well, maybe not cry, but you should definitely fix any dark red. These areas are drawn five times (or more), so just imagine your single device powering five full screens and you should realize how bad this is.

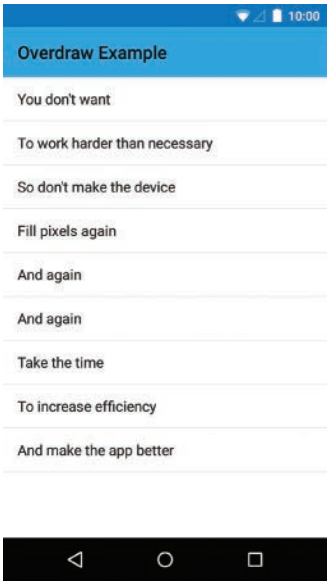

Figure 10.8 Simple app that shouldn't require much work by the device

Figure 10.8 shows a very simple design that is just a list of text. Judging by the simple appearance, it looks like the device has to do minimal work to get these pixels on the display; however, we can turn on the overdraw visualization and see how bad things actually are. Figure 10.9 shows the design tinted by the Show GPU Overdraw option. The full source code for this example is in the OverdrawExample project in the `chapter10` folder.

Figure 10.9 The same app with overdraw being displayed

First, you should know that the theme your app uses should have a window background. This can be a drawable and it is usually a simple color. Android uses this when your app is first being displayed to immediately show some visual indication of what your app should look like even before the views have been inflated. This makes the device feel more responsive because it can respond to the tap even before you've initiated the activity and inflated its views. This window background stays on the screen and everything else is painted on top. That means if you have an activity with a layout that also draws a background, you can very easily draw all pixels on the screen twice and then more times when the views are put on top.

Another common source of overdraw is with items in a `ListView`. The example is not only drawing a white background with the activity's base view right on top of the window background, it is drawing a white background for each list item over that, which is entirely unnecessary. By simply eliminating the extra backgrounds, the overdraw visualization (shown in Figure 10.10) is immediately improved.

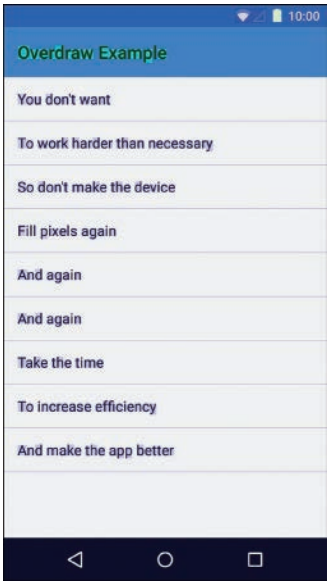

Figure 10.10 Overdraw visualization after eliminating extra backgrounds

Remember that you don't need to eliminate all overdraw, but you should generally minimize it. You should be conscious of what views are drawing in front of and check the overdraw visualization now and then.

Hierarchy Viewer

Hierarchy Viewer is the unsung hero of layout optimization. It hasn't seen a whole lot of change since the early versions, but it still remains a great go-to tool for figuring out what is happening with your layouts. It can be used for simply figuring out why a view doesn't display or to figure

out why a hierarchy is slow to load. It can even output your views as a PSD, allowing you to inspect positioning and colors with precision that is hard to match from simple screenshots.

If your device is running Android 4.0 (or lower) and is unlocked, everything should just work. If it's running 4.0 (or lower) and it is locked, you can use the ViewServer class from Romain Guy (https://github.com/romainguy/ViewServer) in your app (be sure to add the internet permission). If your device is running Android 4.1 or newer, you need to set an environment variable called `ANDROID_HVPROTO` with a value of `ddm`. In Windows, you can open My Computer, Property, Advanced, Environment Variables, and click New to create it. For Mac you'll open `.bash_profile` in your home directory (note that the file starts with a dot, which means it is hidden by default). Add a line that contains `export ANDROID_HVPROTO=ddm` and save the file. Now type source `~/.bash_profile` from the command line (this causes the file to be re-read so that the variable is immediately set). For Linux, you can follow the same steps as for Mac but the file is `.bashrc` in your home directory.

Open Android Device Monitor (under the Tools menu and the Android submenu). The Hierarchy Viewer is a different perspective, so open the Window menu and click the Open Perspective option. Select Hierarchy View and click OK. If you haven't already connected your device and opened the screen you want to inspect, do so now.

On the left side, you should see your device(s) listed. Select it and click the "Load view hierarchy" button (that's the icon next to the refresh button; you can also click the downward-facing triangle and select the option there). If the icon is grayed out, that typically means there is an issue communicating with the device and more details should be available in the console (usually on the bottom right). If you've already followed the directions from two paragraphs ago and it's still gray, you can also try closing out Android Studio (and anything else that might be communicating with the device) and then run Android Device Monitor directly (run `monitor` from the Android SDK tools directory).

Once the view hierarchy has loaded, the left window will show view properties, the center of the screen will be the detailed view hierarchy, the top right will be an overview, and the bottom right is the layout view that lets you see what portion of the screen the selected view is responsible for (the bottom right may be showing the console tab, so just click the Layout View tab). Your screen should look like Figure 10.11.

Each gray box in the tree view (the center window) represents a view. The boxes can have the class type (e.g., `LinearLayout`), the memory address, the ID (e.g., `id/content`), performance indicators, and a view index. The view index shows you the view's position within the parent, where the first child is position 0. The performance indicators are simply colored circles that indicate the time it took to measure the view, the time it took for the layout pass, and the time it took to draw the view. Newer versions of Hierarchy Viewer require you to click the icon with the three circles to obtain the layout times. These indicators on the gray boxes are broken into three groups. If a view is within the fastest 50% of views for the given indicator (e.g., draw time),

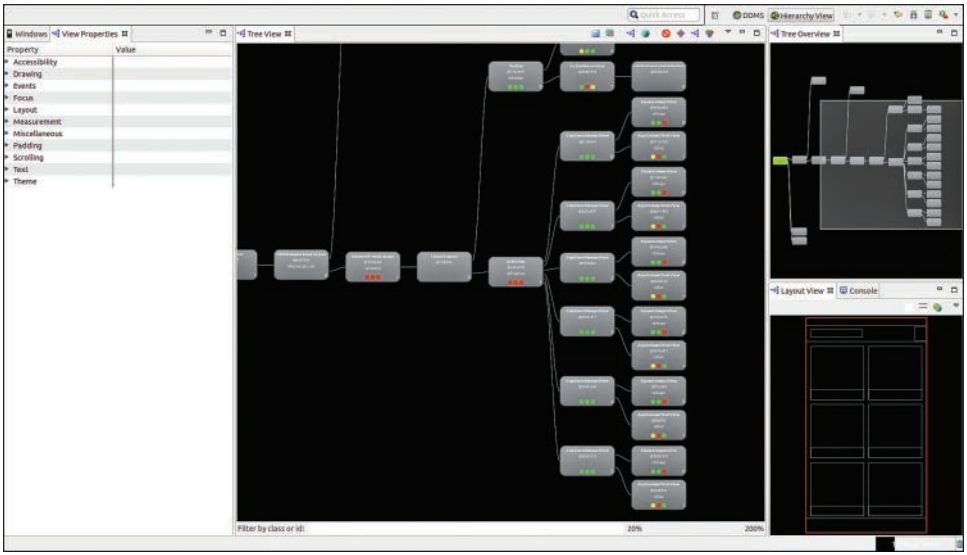

Figure 10.11 After the view hierarchy has been loaded, this is what you should see

that view will be green for that circle. If it's in the 50% of slow views, it will be yellow. If it is the slowest of all the views, it is red. It's important to realize these are relative indicators, so a view hierarchy that is extremely fast and efficient will still have a view with a red indicator for each circle just as an extremely slow hierarchy will.

By clicking a gray box, you can see an image of the view, a count of how many views this view represents (a 1 indicates the view itself; a 2 indicates the view plus a child view), and the exact times for measuring, laying out, and drawing the view. In the view properties (the left window), you can see virtually everything you could want to know about a view. This is extremely helpful when troubleshooting.

Finding Missing Views

One particularly handy use of Hierarchy Viewer is to figure out why a given view isn't showing up. There are many different reasons a view might not appear on the screen, so being able to see all the view's properties in one place plus a visual representation of the views in your hierarchy is incredibly useful. You can quickly see that a view's alpha is 0 or its visibility is set to invisible. You can tell if the view was sized incorrectly or positioned incorrectly. Before Android was blessed with Lint checks, pretty much every developer at one point (or many points) would have a `LinearLayout` and some child views with their widths set to `match_parent`, run the app, and then wonder why only the first child showed up. Simple things like forgetting the default orientation is horizontal can leave you with unexpected results, but Hierarchy Viewer can easily show you where a view is positioned and seeing it on the right edge of the screen

instead of below the previous view is usually enough to get the developer to realize the simple mistake.

Eliminating Unnecessary Views

The biggest benefit Hierarchy Viewer can bring is helping you understand the complexity of your view hierarchy and eliminate extra views. The more complex your hierarchy, the longer your UI thread has to lock up on measuring, laying out, and drawing your views. You should look for views that have only a single child because those are often extraneous views that are easy to remove. You should also look for several sibling `TextView`s because you can often consolidate them (details are later in this chapter). It's a good idea to look for invisible views too. Although they aren't drawn, views that have visibility set to `INVISIBLE` are still measured and even a view that is `GONE` will slow down your view lookups and take up memory. If you sometimes need some views on a given screen but not always, use a `ViewStub` instead of inflating the entire view hierarchy and not drawing it. It's also a good idea to look out for views entirely outside of the screen (such as views that have been animated off the screen); it doesn't do any good to waste processing power on something that will never be seen.

Exporting to PSD

One of the extremely powerful but often overlooked features of Hierarchy Viewer is the ability to export a layout hierarchy as a Photoshop Document (a PSD file). This can be hugely valuable to designers, so make sure they are aware of this functionality. There is a rather forgettable-looking button above the tree view that appears to be three overlapping squares. That's the Capture Layers button. If you do not see it, you can also click the downward-facing triangle to get the list of options and select it from there. The resulting PSD can take a while to be generated, so be patient. If it fails, you will see an error in the console and can try again (occasionally it helps to reconnect the hardware device or restart the emulator).

Because this PSD is not using any advanced features such as layer masks, you can actually open it in GIMP and other tools as well. Along with the techniques discussed earlier in the chapter, this is an excellent method of detecting overdraw.

Exporting to a PSD is a great way for a developer and a designer to speak the same language. The designer can inspect in detail exactly what is going on with a layout by tweaking the layers and then tell the developer which layer has an issue (the layers are named after the view IDs when present, making it extra easy to associate a layer with a view). This also gives the designer the opportunity to make changes to further optimize the design. Perhaps initially a view seemed best at 50-percent opacity, but now the designer can tweak how opaque a view is just like any layer in Photoshop and determine that 40% is actually better.

One thing to note is that the layers are all rasterized. In simplistic terms, the pixels that each view creates are what are actually exported as layers. `TextView`s do not create actual Photoshop text layers, for instance. That also means that if you have a complex view that's

drawing shapes, text, and images, only the resulting pixels are exported, so you can't see what each "layer" of that view looks like.

Custom Fonts

There are times when using a custom font can improve your app. Some apps designed for reading provide additional font choices for users; other apps might use fonts specific to their brand. When deciding on whether to use an additional font, consider how it helps the user experience. Don't include a font just because it's popular or makes the app look different from others; include a font because a usability study has shown that your app is easier to read with the font or because your brand requires it and you want to avoid using images for custom text.

The Roboto font family was built specifically for Android and is the default font for Android 4.0 (Ice Cream Sandwich) and above. Whenever possible, it is the font you should use. You can download the font from the Google design site (http://www.google.com/design/spec/resources/roboto-noto-fonts.html), and it includes multiple variations. In addition to Roboto regular, there is a thin version, light version, medium version, black version, and condensed version (as well as bold and italic versions where applicable).

Because this font was built for Android specifically, it displays very well on a variety of densities and screen types. Many of the most commonly used fonts today were designed for print, which is a very different medium than an electronic display, so some of the fonts that look great on paper do not reproduce as well onscreen. In particular, if you are considering light or thin fonts, be sure to test them on medium- and high-density displays and test them against low-end devices with AMOLEDs (most modern AMOLEDs are reasonably comparable to LCDs, even besting them in some measures, but older and lower quality AMOLEDs like the ones used for the Nexus S have a different subpixel arrangement than a traditional LCD and on top of that have a low enough resolution that the subpixel arrangement can cause display issues for very thin items).

If you do decide to use an alternate font, you need to put it in a directory called `assets` within the root directory of your project. The easiest way to use a custom font in your app is to extend `TextView` to create your own class. Listing 10.10 shows an example.

Listing 10.10 A Custom `TextView` for Displaying a Font

```
public class TextViewRobotoThin extends TextView {

    /**
     * This is the name of the font file within the assets folder
     */
    private static final String FONT_LOCATION = "roboto_thin.ttf";

    private static Typeface sTypeface;

    public TextViewRobotoThin(Context context) {
        super(context);
```

```
        setTypeface(getTypeface(context));
    }

    public TextViewRobotoThin(Context context, AttributeSet attrs) {
        super(context, attrs);
        setTypeface(getTypeface(context));
    }

    public TextViewRobotoThin(Context context, AttributeSet attrs, int
➥ defStyleAttr) {
        super(context, attrs, defStyleAttr);
        setTypeface(getTypeface(context));
    }

    @TargetApi(Build.VERSION_CODES.LOLLIPOP)
    public TextViewRobotoThin(Context context, AttributeSet attrs, int
➥ defStyleAttr, int defStyleRes) {
        super(context, attrs, defStyleAttr, defStyleRes);
        setTypeface(getTypeface(context));
    }

    /**
     * Returns the Typeface for Roboto Thin
     *
     * @param context Context to access the app's assets
     * @return Typeface for Roboto Thin
     */
     public static Typeface getTypeface(Context context) {
        if (sTypeface == null) {
            sTypeface = Typeface.createFromAsset(context.getAssets(),
FONT_LOCATION);
        }
        returnsTypeface;
    }
}
```

At the top of the class is a static string specifying the name of the font file. Each of the normal constructors calls `setTypeface`. A public static method called `getTypeface()` will create the `Typeface` from the font file in the assets directory, if it hasn't already been created, and then return the `Typeface`. This is useful for times when you might access the `Typeface` for other uses (perhaps you do some custom drawing using this `Typeface` elsewhere). By having this public static method, anywhere in your code that needs this custom `Typeface` has one place to go, and you can just change the `FONT_LOCATION` if you need to change the font everywhere in the app.

You can now use this class anywhere you would use an ordinary `TextView`. For instance, you can replace the default "hello world" `TextView` with this in a new project. Figure 10.12 shows how this custom `TextView` looks on an actual device.

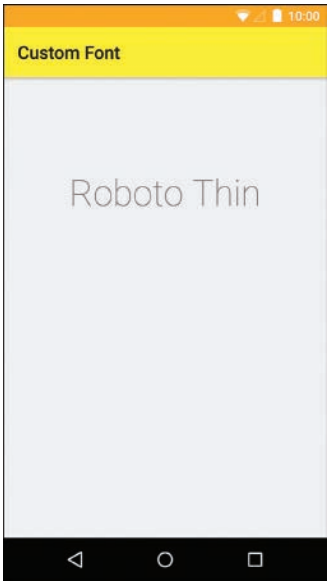

Figure 10.12 The custom `TextView` being displayed on a device

Complex TextViews

`TextView` is an extremely powerful view in Android. Obviously, they're able to display text, but they can also display several styles of text, different fonts or colors, and even inline images, all within a single `TextView`. You can have specific portions of text respond to click events and really associate any object you want with any portion of text. These ranges of text are generically referred to as "spans," as in a span (range) of bold text or a span of subscript.

Existing Spans

Android has a large number of prebuilt spans you can take advantage of. Because you can assign any object as a span, there isn't an actual span class. That's great in that it gives you a huge amount of flexibility, but it also means you have to dig a little to figure out what is supported.

First, you should know about the two main types of spans defined by the interfaces `CharacterStyle` and `ParagraphStyle`. As you can probably guess, these interfaces refer to spans that affect one or more characters and spans that affect entire paragraphs, respectively. Most spans will implement one of these two interfaces (although many implement more than just these). See the following list of built-in spans to get an idea about what is already supported:

- **`AbsoluteSizeSpan`**—A span that allows you to specify an exact size in pixels or density independent pixels.
- **`AlignmentSpan.Standard`**—A span that attaches an alignment (from `Layout.Alignment`).

- **`BackgroundColorSpan`**—A span that specifies a background color (the color behind the text, such as for highlighting).
- **`ClickableSpan`**—A span that has an `onClick` method that is triggered. (This class is abstract, so you can extend it with a class that specifies the `onClick` behavior.)
- **`DrawableMarginSpan`**—A span that draws a `Drawable` plus the specified amount of spacing.
- **`DynamicDrawableSpan`**—A span that you can extend to provide a `Drawable` that may change (but the size must remain the same).
- **`EasyEditSpan`**—A span that just marks some text so that the `TextView` can easily delete it.
- **`ForegroundColorSpan`**—A span that changes the color of the text (basically just called `setColor(int)` on the `TextPaint` object).
- **`IconMarginSpan`**—A span that draws a `Bitmap` plus the specified amount of spacing.
- **`ImageSpan`**—A span that draws an image specified as a `Bitmap`, `Drawable`, `URI`, or resource ID.
- **`LeadingMarginSpan.Standard`**—A span that adjusts the margin.
- **`LocaleSpan`**—A span that changes the locale of text (available in API level 17 and above).
- **`MaskFilterSpan`**—A span that sets the `MaskFilter` of the `TextPaint` (such as for blurring or embossing).
- **`MetricAffectingSpan`**—A span that affects the height and/or width of characters (this is an abstract class).
- **`QuoteSpan`**—A span that puts a vertical line to the left of the selected text to indicate it is a quote; by default the line is blue.
- **`RasterizerSpan`**—A span that sets the `Rasterizer` of the `TextPaint` (generally not useful to you).
- **`RelativeSizeSpan`**—A span that changes the text size relative to the supplied float (for instance, setting a 0.5 float will cause the text to render at half size).
- **`ReplacementSpan`**—A span that can be extended when something custom is drawn in place of the spanned text (e.g., `ImageSpan` extends this).
- **`ScaleXSpan`**—A span that provides a multiplier to use when calling the `TextPaint`'s `setTextScaleX(float)` method. (In other words, setting this to 0.5 will cause the text to be scaled to half size along the X-axis, thus appearing squished.)
- **`StrikethroughSpan`**—A span that simply passes `true` to the `TextPaint`'s `setStrikeThruText(boolean)` method, causing the text to have a line through it (useful for showing deleted text, such as in a draft of a document).
- **`StyleSpan`**—A span that adds bold and/or italic to the text.
- **`SubscriptSpan`**—A span that makes the text subscript (below the baseline).

- **`SuggestionSpan`**—A span that holds possible replacement suggestions, such as for a incorrectly spelled word (available in API level 14 and above).
- **`SuperscriptSpan`**—A span that makes the text superscript (above the baseline).
- **`TabStopSpan.Standard`**—A span that allows you to specify an offset from the leading margin of a line.
- **`TextAppearanceSpan`**—A span that allows you to pass in a `TextAppearance` for styling.
- **`TypefaceSpan`**—A span that uses a specific typeface family (`monospace`, `serif`, or `sans-serif` only).
- **`UnderlineSpan`**—A span that underlines the text.
- **`URLSpan`**—A `ClickableSpan` that attempts to view the specified URL when clicked.

Using Spans for Complex Text

One of the simplest ways to use spans is with the `HTML` class. If you have some HTML in a string, you can simply call `HTML.fromHtml(String)` to get an object that implements the `spanned` interface that will have the applicable spans applied. You can even supply an `ImageGetter` and a `TagHandler`, if you'd like. The styles included in the HTML will be converted to spans so, for example, "b" (bold) tags are converted to `StyleSpan`s and "u" (underline) tags are converted to `UnderlineSpan`s. See Listing 10.11 for a brief example of how to set the text of a `TextView` from an HTML string and enable navigating through and clicking the links.

Listing 10.11 Using HTML in a `TextView`

```
textView.setText(Html.fromHtml(htmlString));
textView.setMovementMethod(LinkMovementMethod.getInstance());
textView.setLinksClickable(true);
```

Another easy method for implementing spans is to use the `Linkify` class. The `Linkify` class allows you to easily create links within text for web pages, phone numbers, email addresses, physical addresses, and so on. You can even use it for custom regular expressions, if you're so inclined.

Finally, you can also manually set spans on anything that implements the `Spannable` interface. If you have an existing `String` or `CharSequence` that you'd like to make `Spannable`, use the `SpannableString` class. If you are building up some text, you can use the `SpannableStringBuilder`, which works like a `StringBuilder` but can attach spans. To the untrained eye, the app in Figure 10.13 is using two `TextView`s and an `ImageView`, but it actually has just a single `TextView`. See Listing 10.12 to understand how you can do this with one `TextView` and a few spans.

Figure 10.13 An app that seemingly uses more views than it really does

Listing 10.12 Using Spans with a `SpannableStringBuilder`

```
final SpannableStringBuilder ssb = new SpannableStringBuilder();
final int flag = Spannable.SPAN_EXCLUSIVE_EXCLUSIVE;
int start;
int end;

// Regular text
ssb.append("This text is normal, but ");

// Bold text
start = ssb.length();
ssb.append("this text is bold");
end = ssb.length();
ssb.setSpan(new StyleSpan(Typeface.BOLD), start, end, flag);

// Inline image
ssb.append('\n');
start = end + 1;
ssb.append('\uFFFC'); // Unicode replacement character
end = ssb.length();
ssb.setSpan(new ImageSpan(this, R.mipmap.ic_launcher), start, end, flag);

// Stretched text
start = end;
ssb.append("This text is wide");
end = ssb.length();
```

```
ssb.setSpan(new ScaleXSpan(2f), start, end, flag);

// Assign to TextView
final TextView tv = (TextView) findViewById(R.id.textView);
tv.setText(ssb);
```

RecyclerView

If you've been doing Android development for some time, you've probably come across issues with `ListView`. For most uses `ListView` is great. It handles everything you need and it is efficient; however, there are times when `ListView` doesn't do everything you want. For instance, animating views within `ListView` has a lot of challenges and there isn't a good way to change how the items are laid out.

To solve these issues, `RecyclerView` was created. Its name is a reference to the process of view recycling that we've discussed previously where a view that goes off screen can then be reused for a new view coming onto the screen. `RecyclerView` works similarly to `ListView` but with a more modular architecture. For instance, instead of assuming that you want your items vertically stacked, you supply a layout manager that handles determine how to lay out the items. Given that `RecyclerView` can do everything that `ListView` can, you might be wondering when you should still use `ListView`. The simple answer is to use `ListView` unless you encounter one of its limitations. Contrary to some opinions, `ListView` isn't dead and you shouldn't change all your `ListView`s over to `RecyclerView`s without reason. If you find that `ListView` doesn't work for you, that's the time to switch over to `RecyclerView`.

If you want a good starting point for playing around with `RecyclerView`, check out the sample. From Android Studio, open the File menu and select Import Sample and then type "Recycler View" in the search box. This will let you quickly set up a project that uses `RecyclerView` so that you can see how it works and experiment with it.

Layout Manager

To tell `RecyclerView` how to arrange the views, you need a layout manager. It's responsible for measuring and positioning views as well as handling the view recycling policy. Fortunately, there are some built in ones for the typical use cases. You'll commonly use `LinearLayoutManager`, which can give you a `ListView`-like display but also supports horizontal arrangement. You may also use `GridLayoutManager` or `StaggeredGridLayoutManager` for times when you want grids of items rather than just rows or columns.

Adapter

Like `ListView`, `RecyclerView` uses an adapter to supply its views, but it works a little differently (also note that it's `RecyclerView.Adapter` rather than `android.widget.`

Adapter). You commonly have to use the view holder pattern with adapters for ListView as described earlier in the chapter to avoid repeated calls to findViewById, but that pattern is built in to RecyclerView with an actual ViewHolder class. Rather than the getView method, this adapter has two methods. First, onCreateViewHolder is expected to create any views that are necessary, but it returns a new ViewHolder instance that references those views. Second, onBindViewHolder is expected to set all the views up for a given position using the ViewHolder instance.

Note that a key difference between AdapterView and RecyclerView is how you listen to clicks. Rather than assigning a single onItemClickListener, RecyclerView just uses the regular onClickListener and related classes. Typically any necessary listeners are set in onCreateViewHolder and the position or object(s) they care about are updated in onBindViewHolder.

Item Animator

Whenever the state of the RecyclerView changes, it needs to know how to animate the change and that's where the ItemAnimator abstract class comes in. A concrete implementation allows you to control the animations that are used to visualize these changes, so that your users can easily understand what happened. By default, RecyclerView uses the aptly named DefaultItemAnimator, but you can provide a custom implementation to precisely control the behavior.

Item Decoration

RecyclerView has one more trick up its sleeve and that's the concept of item decoration, which is based on the ItemDecoration abstract class. The idea is simple: sometimes you need to draw more than just the views that are displaying your data. For instance, you might draw dividers between items or visually group a section of items. ItemDecoration has both onDraw and onDrawOver methods. The former draws before the views are drawn (such as for backgrounds) and the latter draws after (meaning it draws in front of them). You can add multiple ItemDecoration implementations to your RecyclerView to get the effect you want, but you don't have to use any.

One additional feature ItemDecoration has is the ability to modify the position of views. By overriding getItemOffsets, you can adjust positions as needed, which allows you to manipulate the display in powerful ways such as visually clustering content based on proximity or spacing.

Summary

In this chapter, you learned a wide variety of topics that help advanced developers make even better apps. Using Systrace will allow you to get performance details that help you make informed decisions on how you can improve your app and help you identify issues that are

easily missed like decoding an image twice. You saw multiple techniques for improving the efficiency of working with images from pre-sizing to background loading. Additional tools and techniques for performance improvements followed before getting into advanced uses of existing views by way of spans and the flexible `RecyclerView`.

In the next chapter, you'll dive right into the world of drawing in Android. You'll learn how to draw images and text as well as how to apply filters and compositing.

This page intentionally left blank

INDEX

A

abortAnimation(), 370
AbsListView class, 284
AbsoluteLayout, 62
AbsoluteSizeSpan, 295
AccelerateDecelerateInterpolator class, 239
AccelerateInterpolator class, 239
accessible vocabulary, 182
action bar. *see* app bar
ActionBarCompat library, 70
activity transitions, 255–259
Adapter class, 60–61
AdapterViewAnimator, 62
AdapterView class, 57
  interfaces for, 61
AdapterViewFlipper, 62
add, PorterDuff image compositing mode, 322
Adobe Illustrator, 118, 168–169
AlignmentSpan.Standard, 295
alpha, View class attribute, 28
Amazon Appstore, 401
ambient light, 171
AnalogClock, 45–46
Android Asset Studio, 390–395
Android design website, 13
android:state_activated, 89
android:state_checkable, 89
android:state_checked, 89
android:state_enabled, 89
android:state_focused, 89
android:state_hovered, 90
android:state_pressed, 90
android:state_selected, 90
android:state_window_focused, 90
angle, ShapeDrawable class attribute, 94
AnimatedVectorDrawable class, 98–100
animations, 104
  form errors, 242–246
  icons, 246–251
  and Material Design, 11–12
  property. *see* property animations
  purpose of, 232
  view, 232–233
AnimatorListener class, 235
animator (qualifier), 76
anim (qualifier), 76
AnticipateInterpolator class, 239
AnticipateOvershootInterpolator class, 239
app bar, 21–22
AppCompat library, 70–71
Apple, 170
application description, Google Play, 388
application icon, Google Play, 389–395
apps
  Camera app, 15
  Contacts app, 14–15
  Gallery app, 15
  Paper Camera, 400
  Permission Informant, 175
  promoting apps, 400–401
AppWidgetHostView, 62
arrays, 103
attributes, ShapeDrawable class, 94–95
  angle, 94
  bottom, 95
  bottomLeftRadius, 94
  bottomRightRadius, 94
  centerColor, 94
  centerX, 94
  centerY, 94
  color, 95
  dashGap, 95
  dashWidth, 95
  endColor, 94
  gradientRadius, 95
  height, 95
  innerRadius, 94
  innerRadiusRatio, 94
  left, 95
  radius, 94
  right, 95

attributes, ShapeDrawable class (*continued*)
shape, 94
startColor, 95
thickness, 94
thicknessRatio, 94
top, 95
topLeftRadius, 94
topRightRadius, 94
type, 95
useLevel, 94
width, 95
attributes, TextView class, 36–38
drawable, 36
ellipsize, 36
fontFamily, 36
gravity, 36
hint, 36
inputType, 36
lines, 37
lineSpacing, 37
maxLines, 37
minLines, 37
shadowColor, 37
shadowDx, 37
shadowDy, 37
shadowRadius, 37
text, 37
textColor, 37
textIs, 37
textSize, 37
textStyle, 37
typeface, 37
attributes, VectorDrawable class, 97–98
attributes, View class
alpha, 28
background, 29
content, 29
duplicate, 29
focusable, 29
id, 29
importantFor, 29
minHeight, 29
minWidth, 29
padding, 30
visibility, 30
attrs.xml, 409
AutoCompleteTextView, 42
available height, 78
available width, 78

B

back button, 19
background, View class attribute, 28
BackgroundColorSpan, 296
backgrounds, 39–40
Balsamiq, 119
bending to user, core principles, 19
better text drawable, 308–310
Bitmap class, 311, 338–339
BitmapShader, 325–328
bottom, ShapeDrawable class attribute, 95
bottomLeftRadius, ShapeDrawable class attribute, 94
bottomRightRadius, ShapeDrawable class attribute, 94
BounceInterpolator class, 239
breaking comps into views, 200–201
brightness, 173
Button class, 39

C

cache, images, 277–283
CalendarView, 42
Camera app, 15
Canvas class, 305
capitalization, text, 180
CardView library, 71–72
centerColor, ShapeDrawable class attribute, 94
centerX, ShapeDrawable class attribute, 94
centerY, ShapeDrawable class attribute, 94
change log, Google Play, 389
CharSequence class, 236–238
CheckBox, 42
CheckedTextView, 42
Chronometer, 46
circular reveal transitions, 259–262
clear, PorterDuff image compositing mode, 316
ClickableSpan, 296
ClipDrawable class, 92–93
CMYK (cyan, magenta, yellow, and black), 172
color, ShapeDrawable class attribute, 95
Color (Adobe's tool), 177
color blindness, 177–178
ColorMatrixColorFilter class, 314–315
color (qualifier), 76

colors, 103, 171–178
  cultural meanings, 175–176
  filters, 313–325
  selection of, 174–178
complex text, spans for, 297–299
components, Android, 20–23
  app bar, 21–22
  FAB, 23
  navigation drawer, 23
  notifications, 21
  system bars, 20
  tabs, 22–23
ComposeShader, 325
CompoundButton, 42
comps, breaking into views, 200–201
computeScroll(), 365
configuration support, 113–114
Contacts app, 14–15
content, View class attribute, 29
content pieces, wireframes, 122–123
context.getResources(), 355
Contrast Analyser, 179
contrast ratio, text, 179
convertView parameter, 59, 61
core principles
  bending to user, 19
  do one thing well, 14–15
  easy but powerful, 17–18
  platform consistency, 18–19
  visuals, 17
  work well with others, 15–16
custom drawables, creation of, 304–305
custom fonts, 181–182, 293–295
custom views
  attributes, 409–416
  bitmap creation, 338–339
  creating initial custom view files, 352–357
  creation of, 334–347, 352–385
  drawing, 360–365
  layout, 337–338
  measurement, 336–337, 357–360
  saving and restoring state, 340–347
CycleInterpolator class, 239

D

darken, PorterDuff image compositing mode, 320–321
dashGap, ShapeDrawable class attribute, 95
dashWidth, ShapeDrawable class attribute, 95
DatePicker, 42
Daydream feature, 6
DecelerateInterpolator class, 239
defaults, specifying, 81
deferred rendering, 286–287
density, 79, 81–83
design
  colors, 171–178
  error cases, 184–185
  graphical, wireframes and, 168
  icons, 184
  lighting, 171
  methods, 108–110
  navigation, 184
  procedural steps, 185–189
  text. *see* text
  transitions, 184
designers, working with, 192–193
design library, 72
design website, 13
detail pages, wireframes, 123–125
developers, 192
Device Art Generator, 401
device support, 113–114
DialerFilter, 62
DigitalClock, 46
dimensions, 104
display
  images, 39–41
  text, 34–39
dots per inch (DPI), 31, 81
  determining, 407
drawable, TextView class attribute, 36, 76
Drawable class, 304–305, 356–357
DrawableMarginSpan, 296
drawables
  custom, 304–305
  XML drawables. *see* XML drawables
draw(Canvas), 304
drawing views, 333, 360–365
draw9patch tool, 195
DST, PorterDuff image compositing mode, 316–317
DST_ATOP, PorterDuff image compositing mode, 320
DST_IN, PorterDuff image compositing mode, 318

DST_OUT, PorterDuff image compositing mode, 318
DST_OVER, PorterDuff image compositing mode, 318–319
Duarte, Matias, 6
Duff, Tom, 315
duplicate, View class attribute, 29
DynamicDrawableSpan, 296

E

EasyEditSpan, 296
EdgeEffect class, 351
EditText, 38, 404
eliminating, overdraw, 286–289
ellipsize, TextView class attribute, 36
endColor, ShapeDrawable class attribute, 94
error cases, design, 184–185
events, listening to, 47–48
exit button, 19
ExpandableListView class, 59
external libraries, 114
extra, extra, extra high dots per inch (XXXHDPI), 32, 81, 198
extra, extra high dots per inch (XXHDPI), 32, 81
ExtractEditText, 46
extra high dots per inch (XHDPI), 32, 81

F

FAB, 23
FastOutLinearInInterpolator class, 239
FastOutSlowInInterpolator class, 240
feature graphic, Google Play, 397–399
feedback, real user, 162–164
filters, colors, 313–325
flat design, 170
flowcharts, 114
focusable, View class attribute, 29
focused state, 17
fontFamily, TextView class attribute, 36
fonts
  custom, 181–182, 293–295
  Roboto font family, 293
  TextView class, 293
ForegroundColorSpan, 296
FragmentBreadCrumbs, 62
fragments, 64
  and activities, 130
  lifecycle of, 64–65
  passing data to, 65–66
  problems associated with, 69–70
  prototypes, creation of. *see* prototypes, creation of
  tab, 142–155
  talking to activity from, 66–69
  transactions, 69
FrameLayout class, 52
full screen mode, 404–405

G

Gallery app, 15
Gallery class, 60
garbage collection, controlling, 283–284
GestureDetector class, 350
GestureOverlayView, 62
getBounds(), 304
getCurrVelocity(), 385
getIntrinsicHeight(), 304
getIntrinsicWidth(), 305
getOpacity(), 304
getScaledMinimumFlingVelocity(), 355
getScrollX(), 365
GIFs, 83
Gimp, 168
GLSurfaceView, 46
goals, 110–111
  device and configuration support, 113–114
  product, 112–113
  user, 111–112
  user personas, 112
Google Play, 387
  application description, 388
  application icon, 389–395
  change log, 389
  feature graphic, 397–399
  promoting apps, 400–401
  promotional graphic, 399–400
  screenshots, 395–397
  video (YouTube), 400
Google Plus, 16
gradientRadius, ShapeDrawable class attribute, 95
gradient shader, 325, 328–330

graphical design
styles, 169–170. *see also* specific styles
tools for, 168–169
wireframes and, 168
graphics
feature graphic, Google Play, 397–399
promotional. *see* promotional graphics
`gravity`
`LinearLayout` class attribute, 52–53
`TextView` class attribute, 36
`GridLayout`, 63
GridLayout library, 72
`GridView` class, 59

H

hardware keyboard type, 80
HDPI (high dots per inch), 32, 81
`height`, `ShapeDrawable` class attribute, 95
Hierarchy Viewer, 289–291
eliminating unnecessary views, 292
exporting Photoshop Document, 292–293
missing views, finding, 291–292
high dots per inch (HDPI), 32, 81
high-level flow, 114–116
`hint`, `TextView` class attribute, 36
history, Android design, 6
home screen, 7
Honeycomb, 6, 233
full screen mode, 405
`HorizontalIconView` class, 352–355
`HorizontalScrollView` class, 63
hot spot, 171
HSB (hue, saturation, and brightness), 172
vs. RGB, 173–174
`HTML` class
spans, 297–299
hue, 172

I

Ice Cream Sandwich, 6
`IconMarginSpan`, 296
icons
animations, 246–251
design, 184
raster, animating, 250–251
vector, animating, 246–250
`id`, `View` class attribute, 29
IDs, 104–105
Illustrator, 118, 168–169
`ImageButton` class, 41
image files, 83–84. *see also* raster images; vector images
images
alternate sizes, generating, 198–199
availability and size, 183
cache, 277–283
display, 39–41
nine-patch, 84–86, 195–198
optimization, 272–283
right sizes for, 273–277
round, shaders for, 325–328
shrinking, 199, 272–273
working with, 310–313
XML drawables. *see* XML drawables
`ImageSpan`, 296
`ImageSwitcher`, 63
`ImageView` class, 40–41
`importantFor`, `View` class attribute, 29
ink, as Material Design component, 10–11
Inkscape, 118
`innerRadius`, `ShapeDrawable` class attribute, 94
`innerRadiusRatio`, `ShapeDrawable` class attribute, 94
`InputMethodManager`, dismissing software keyboard, 404
`inputType`, `TextView` class attribute, 36
`InsetDrawable` class, 92
interaction, and Material Design, 11–12
interpolators, 232–233, 236
`IntEvaluator` class, 235
`ItemAnimator` class, 300
`ItemDecoration` class, 300

J

jank
identifications, 264–265
Systrace and, 265–271
Jelly Bean, 6
JPEGs, 83, 272–273

K

kerning, 180
keyboard availability, 80

keyboards, software keyboard (dismissing), 404
KeyboardView, 46
key frames, 240–241
KitKat, 256

L

language, 77
language direction, 77
layer list, XML drawables, 87–88
LayoutParams, RelativeLayout class, 56–57
layout (qualifier), 76
layouts, 28
  views, 333, 337–338
LayoutTransition class, 252
LDPI (low dots per inch), 32, 81
leading, 180
LeadingMarginSpan.Standard, 296
leanback library, 73
left, ShapeDrawable class attribute, 95
level list, XML drawables, 91
LevelListDrawable class, 91
lifecycle, of fragments, 64–65
lighten, PorterDuff image compositing mode, 320
lighting, 171
LightingColorFilter class, 313
LinearGradient, 325
LinearInterpolator class, 240
LinearLayout class, 52–55
LinearLayoutManager class, 299
LinearOutSlowInInterpolator class, 240
lines, TextView class attribute, 37
line spacing, 180
lineSpacing, TextView class attribute, 37
listeners, 235
ListView class, 58–59, 284, 299
LocaleSpan, 296
Lollipop, 7
long press gesture, 25
low dots per inch (LDPI), 32, 81
LruCache class, 277
LRU (least-recently-used) cache, 277
LTR (left-to-right) layout, 32–34

M

MaskFilterSpan, 296
match_parent (layout parameter), 31, 55
Material Design, 8–13, 170
  and app bar, 22
  colors for, 173–174
  interaction and animation, 11–12
  metrics and alignment, 13
  mistakes made by designers/developers, 24–25
  overview, 8–11
  typography, 12–13
maxLines, TextView class attribute, 37
MDPI (medium dots per inch), 32, 81
measureHeight(int), 358
measurement, views, 332–333, 336–337, 357–360
MeasureSpec, 333
measureWidth(int), 358, 359
mEdgeEffectLeft, 356, 370
mEdgeEffectRight, 356, 370
MediaController, 63
MediaRouteButton, 46
MediaRouter library, 73
medium dots per inch (MDPI), 32, 81
menu key, 24–25
menu (qualifier), 76
menus, 105
methods, design
  common, 108
  User-Centered Design, 108–110
MetricAffectingSpan, 296
Metro (design language), 170
Microsoft, 170
minHeight, View class attribute, 29
Minimalism, 170
minLines, TextView class attribute, 37
minWidth, View class attribute, 29
mipmap (qualifier), 76
mistakes, made by designers/developers
  long press gesture, 25
  menu key, 24–25
  notification icons, 25
  styles from other platforms, 25
mobile country code, 77
mobile network code, 77
modes, PorterDuff image compositing, 316–325
  add, 322
  clear, 316
  darken, 320–321
  DST, 316–317

DST_ATOP, 320
DST_IN, 318
DST_OUT, 318
DST_OVER, 318–319
lighten, 320
multiply, 322–323
overlay, 323
screen, 323
SRC, 316–317
SRC_ATOP, 320
SRC_IN, 318
SRC_OUT, 318
SRC_OVER, 318–319
XOR, 323–325
MotionEvent.ACTION_CANCEL, 371
MotionEvent.ACTION_POINTER_UP, 371
MotionEvent.ACTION_UP, 370–371
MotionEvent class, 350–351, 369
MultiAutoCompleteTextView, 44
multiple devices, support for, 23–24, 125–126
multiply, PorterDuff image compositing mode, 322–323

N

naming conventions, wireframes, 127
navigation
design, 184
wireframes, 119–122
navigation drawer, 23, 134–136
navigation key availability, 80
network connection, 408
Nexus 5, 268, 273
night mode, 79
nine-patch images, 84–86, 195–198
normal state, 17
notification icons, 25
notifications, 21
NumberPicker, 44

O

ObjectAnimator class, 234
Omnigraffle, 118
onAbsorb(int), 365
onActivityCreated(Bundle), 64
onAttach(Activity), 64
onBoundsChange(Rect), 305
OnClickListener, 48
onCreate(Bundle), 64
onCreateView(LayoutInflater, ViewGroup, Bundle), 64
onDestroy(), 65
onDestroyView(), 65
onDetach(), 65
OnDragListener, 48
onDraw(Canvas), 355
OnFocusChangeListener, 48
OnGenericMotionListener, 48
OnHoverListener, 48
onKeyDown(int, KeyEvent), 351
OnKeyListener, 48
onLevelChange(int), 305
OnLongClickListener, 48
onMeasure(int, int), 360
onOverScrolled(int, int, boolean, boolean), 366
onPause(), 65
onResume(), 64
onScrollChanged(x, 0, oldX, 0), 365
onStart(), 64
onStateChange(int[]), 305
onStop(), 65
onTouchEvent(MotionEvent), 369, 371
OnTouchListener, 48
onViewStateRestored(Bundle), 64
optical bounds, 197–198
optimization, images, 272–283
orientation, 79
orientation, LinearLayout class, 54, 55
overdraw, eliminating, 286–289
overlay, PorterDuff image compositing mode, 323
overScrollBy, 366, 370
OverScroller class, 351
OvershootInterpolator class, 240

P

padding, View class attribute, 30
PagerTabStrip, 63
PagerTitleStrip, 63
Paint class, 305
Palette library, 73
paper, as Material Design component, 8–9

Paper Camera app, 400
PathInterpolator class, 240
patterns, ViewHolder class, 284–286
performance, techniques to improve
  eliminating overdraw, 286–289
  garbage collection, controlling, 283–284
  view holder pattern, 284–286
Permission Informant app, 175
photo filters, 17
Photoshop, 168, 177–178
Photoshop documents (PSD), exporting, 292–293
Pixelmator, 168
platform consistency, core principles, 18–19
platform version, 80
PNGs, 83, 273
  compression, 84
Porter, Thomas, 315
PorterDuff image compositing, 315–316
  modes, 316–325. *see also* modes, PorterDuff image compositing
postInvalidateOnAnimation(), 385
pressed state, 17
primary non-touch navigation method, 80
product goals, 112–113
ProgressBar, 46
Project Butter, 6
promoting apps, Google Play, 400–401
promotional graphics
  Google Play, 399–400
property animations, 233–235
  control, 235–241. *see also* specific controls
prototypes, creation of, 131–160
  navigation drawer, 134–136
  tabs, 132–134
  tool details, 155–160
  tool representation, 137–142
prototypes, evaluation of, 160
  dynamic goal, 162
  open-ended exploration, 161
  real user feedback, 162–164
  specific goal, 161–162
  working with users, 161

Q

QuickContactBadge, 46
QuoteSpan, 296

R

RadialGradient, 325
RadioButton, 44
RadioGroup, 44
radius, ShapeDrawable class attribute, 94
raster icons, animating, 250–251
raster images, 83
RasterizerSpan, 296
RatingBar, 44
raw (qualifier), 76
R class, 30–31
  resource qualifiers, 80–81
R.drawable.header, 80, 81
RecyclerView.Adapter class, 299–300
RecyclerView class, 299–300
RecyclerView library, 73
region, 77
RelativeLayout class, 55–56
RelativeSizeSpan, 296
rendering, deferred, 286–287
ReplacementSpan, 296
resource qualifiers, 76–81
  list of, 77–80
RGB (red, green, and blue)
  vs. HSB, 173–174
right, ShapeDrawable class attribute, 95
RippleDrawable class, 100–101
Roboto, 12, 13
Roboto font family, 293
rounded corners, specifying, 95
round images, shaders for, 325–328
RSSurfaceView, 47
RSTextureView, 47
RTL (right-to-left) layout, 32–34

S

Samsung, 34
saturation, 172–173
ScaleDrawable class, 93
scaleType attribute, values for, 40–41
ScaleXSpan, 296
scene transitions, 252–255
screen, PorterDuff image compositing mode, 323
screen aspect, 79
screens, keeping on, 405
screenshots, Google Play, 395–397

screen size, 78
  determining, 406–407
Scroller class, 351
ScrollerCompat class, 351
ScrollView, 63
SearchView, 63
SeekBar, 44
setAlpha(int), 304
setColorFilter(ColorFilter), 304
setDrawables(ListDrawable), 356–357
setWillNotDraw(false), 355
setWillNotDraw method, 334
shaders, 325–330
  gradient, 325, 328–330
  round images, 325–328
shadowColor, TextView class attribute, 37
shadowDx, TextView class attribute, 37
shadowDy, TextView class attribute, 37
shadowRadius, TextView class attribute, 37
shadows
  in Material Design, 9–10
  text, 180–181
shape, ShapeDrawable class attribute, 94
ShapeDrawable class, 93–96
sharing, 15–16
Show GPU Overdraw option, 287, 288
shrinking images, 199, 272–273
simple text drawable, 306–308
sizes, text, 180
Sketch tool, 118–119
skeuomorphic design, 11
skeuomorphism, 169
slicing, 193
  easy, 193–194
  generating alternate sizes, 198–199
  nine-patch images, 195–198
SlidingDrawer, 63
smallest width, 78
software keyboard, dismissing, 404
Space, 47
spacing, text, 180
spans
  for complex text, 297–299
  existing, 295–297
Spinner, 44
Spinner class, 60
SRC, PorterDuff image compositing mode, 316–317
SRC_ATOP, PorterDuff image compositing mode, 320
SRC_IN, PorterDuff image compositing mode, 318
SRC_OUT, PorterDuff image compositing mode, 318
SRC_OVER, PorterDuff image compositing mode, 318–319
StackView, 63
standard icons, 184
startColor, ShapeDrawable class attribute, 95
state list, XML drawables, 88–91
StateListDrawable class, 88
StrikethroughSpan, 296
strings, 102–103
styles
  text, 180
  themes and, 199–200
StyleSpan, 296
SubscriptSpan, 296
SuggestionSpan, 297
SuperscriptSpan, 297
supporting multiple devices, 23–24, 125–126
support library
  annotations library, 73–74
  AppCompat library, 70–71
  CardView library, 71–72
  design library, 72
  GridLayout library, 72
  leanback library, 73
  MediaRouter library, 73
  Palette library, 73
  RecyclerView library, 73
SurfaceView, 47
SweepGradient, 325
Switch, 44
system bars, 20
Systrace tool, 265–271

T

TabHost, 63
TableLayout, 63
TableRow, 63
tabs, 22–23, 132–134
TabStopSpan.Standard, 297
TabWidget, 63

testing, across device types, 228–229
text
  better drawable, 308–310
  capitalization, 180
  complex, spans for, 297–299
  contrast ratio, 179
  design, 178–182
  display, 34–39
  fonts, 181–182
  shadows, 180–181
  simple drawable, 306–308
  sizes, 34, 180
  spacing, 180
  styles, 180
  varying lengths, 183
`text, TextView` class attribute, 37
`TextAppearanceSpan`, 297
`TextClock`, 47
`textColor, TextView` class attribute, 37
`textIs, TextView` class attribute, 37
`textSize, TextView` class attribute, 37
`textStyle, TextView` class attribute, 37
`TextSwitcher`, 63
`TextureView`, 47
`TextView` class, 35–38
  attributes, 36–38
  existing spans, 295–297
  fonts, 293
  HTML, 297–299
`thickness, ShapeDrawable` class attribute, 94
`thicknessRatio, ShapeDrawable` class attribute, 94
3x Rule, 183–184
`TimeInterpolator` interface, 236–238
`TimePicker`, 45
`ToggleButton`, 45
`Toolbar` class, 62
`top, ShapeDrawable` class attribute, 95
`topLeftRadius, ShapeDrawable` class attribute, 94
`topRightRadius, ShapeDrawable` class attribute, 94
touch input
  handling, 369–385
  preparation for, 365–369
  and views, 350–351
touchscreen type, 79
`TransitionDrawable` class, 91–92
transitions, 252
  activity, 255–259
  circular reveal, 259–262
  design, 184
  scene, 252–255
transparency, 183–184
`type, ShapeDrawable` class attribute, 95
type evaluators, 235–238
`typeface, TextView` class attribute, 37
`TypefaceSpan`, 297
typography, 12–13

U

UCD (User-Centered Design), 108–110
UI mode, 79
UI threads, 408–409
`UnderlineSpan`, 297
up indicator, 125
`URLSpan`, 297
`useLevel, ShapeDrawable` class attribute, 94
User-Centered Design (UCD), 108–110
user goals, 111–112
user personas, 112

V

`ValueAnimator` class, 233–234, 235
`values` (qualifier), 76
`VectorDrawable` class, 96
  attributes for, 97–98
vector icons, animating, 246–250
vector images, 84
`VideoView`, 47
video (YouTube), 400
view animations, 232–233
`ViewAnimator`, 63
`View` class, attributes. *see* attributes, `View` class
`ViewFlipper`, 63
`ViewGroup` class, 52, 360
`ViewHolder` class, patterns, 284–286
`ViewPager` class, 61
`ViewPropertyAnimator` class, 241–242
views, 28
  breaking comps into, 200–201

custom. *see* custom views
dimensions, 31–34
drawing, 333
for gathering user input, 42–45
general concepts behind, 332
IDs, 30
layout, 333
measurement, 332–333
missing, finding, Hierarchy Viewer and, 291–292
save and restore state, 334
touch input and, 350–351
unnecessary, eliminating, 292
`ViewSwitcher`, 63
`visibility`, `View` class attribute, 30
visuals, core principles, 17

W

WCAG (Web Content Accessibility Guidelines), 179
website, Android design, 13
`WebView`, 47
`weight`, `LinearLayout` class attribute, 53
`width`, `ShapeDrawable` class attribute, 95
wireframes, 117–119
content pieces, 122–123
detail pages, 123–125
and graphical design, 168
naming conventions, 127
navigation, 119–122
supporting multiple devices, 125–126
Wireframe Sketcher, 119
woodworking app, 178
development of, 201–228
main screen, 201–211
tool details activity, 224–228
tool list, 211–224
`wrap_content` (layout parameter), 31

X

XHDPI (extra high dots per inch), 32, 81
XML drawables, 86–87
`AnimatedVectorDrawable` class, 98–100
`ClipDrawable` class, 92–93
`InsetDrawable` class, 92
layer list, 87–88
level list, 91
`RippleDrawable` class, 100–101
`ScaleDrawable` class, 93
`ShapeDrawable` class, 93–96
state list, 88–91
tiles, 194
`TransitionDrawable` class, 91–92
`VectorDrawable` class, 96–98
`xml` (qualifier), 76
`XOR`, PorterDuff image compositing mode, 323–325
XXHDPI (extra, extra high dots per inch), 32, 81
XXXHDPI (extra, extra, extra high dots per inch), 32, 81, 198

Z

`ZoomButton`, 47
`ZoomControls`, 63