Kim Topley

Updated for JavaFX 1.3

JavaFX™ Developer's Guide

Developer's Library

JavaFX™ Developer's Guide

Library of Congress Cataloging-in-Publication Data

Topley, Kim.
JavaFX developer's guide / Kim Topley.
p. cm.
Includes bibliographical references and index.
ISBN 978-0-321-60165-0 (pbk. : alk. paper) 1. Java (Computer program language) 2. JavaFX (Electronic resource) 3. Graphical user interfaces (Computer systems) 4. Application software—Development. 5. Internet programming. I. Title.
QA76.73.J38T693 2010
005.13'3—dc22

2010010696

Text printed in the United States on recycled paper at Edwards Brothers in Ann Arbor, Michigan.
First Printing, October 2010

ISBN 13: 978-0-321-60165-0
ISBN 10: 0-321-60165-3

Trademarks

Warning and Disclaimer

Bulk Sales

Pearson offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales. For more information, please contact U.S. Corporate and Government Sales, 1-800-382-3419, corpsales@pearsontechgroup.com.

For sales outside of the United States, please contact: International Sales, international@pearsoned.com.

Visit us on the web: informit.com/aw

For information regarding permissions, write to: Pearson Education, Inc., Rights and Contracts Department, 501 Boylston Street, Suite 900, Boston, MA, 02116;
Fax: (617) 671-3447.

Acquisitions Editor
Greg Doench

Development Editor
Michael Thurston

Managing Editor
John Fuller

Full-Service Production Manager
Julie B. Nahil

Copy Editor
Keith Cline

Indexer
Rebecca Salerno

Proofreader
Apostrophe Editing Services

Publishing Coordinator
Michelle Housley

Book Designer
Gary Adair

Composition
Jake McFarland

Contents at a Glance

IV: Miscellaneous

Table of Contents

III: User Interfaces with JavaFX

Preface

The official launch of the Java programming language coincided with huge public interest in the World Wide Web. Home computers were becoming affordable, and large numbers of homes were connected to the Internet, mostly using slow, dial-up lines (remember those?), and Netscape Navigator was by far the most popular web browser. In 1994, a version of this browser was shipped with a plug-in that allowed Java applets to be embedded in HTML web pages. Applets brought motion and dynamic content to what had formerly been a mainly static World Wide Web. So great was the impact that a bouncing-head animation, which was actually a Java applet, was shown on an evening news broadcast in the United Kingdom. It seemed that applets and the Java programming language were set for a bright future.

The 1.0 release of the Java Development Kit (JDK) included the compiler together with a relatively small set of class libraries that provided mainly I/O and networking facilities and a primitive user interface toolkit called AWT (Abstract Window Toolkit). One of the most novel things about Java and AWT was that they allowed the programmer to write an application that would run unchanged on both Microsoft Windows and UNIX. The platform became more robust with the release of JDK 1.1, which for a long time was the standard Java platform and was used to write both applets and free-standing applications, deployed on corporate intranets.

Although Java was born as a desktop technology, it didn't stay that way for very long. Novel though it was, there were problems with the applet programming model as implemented in JDK 1.1. The most obvious was speed—both of delivery and of execution. Before the arrival of Java Archive (JAR) files, an applet's class files were hosted on a website and downloaded individually, on demand. This meant that anything other than a very simple applet was slow to start and would be prone to freezing during execution if a new class file had to be fetched. Furthermore, the early versions of the Java Virtual Machine (JVM) were not well optimized, especially when it came to garbage collection, which would often cause execution to appear to be suspended for a noticeable time.

Neither was AWT a comprehensive toolkit—there were very few components, meaning that an applet or application either had to be very basic or its author had to invest considerable time and effort to write and debug custom components. Unfavorable comparisons were being made with native applications and this continued even with the release of Swing, a more comprehensive user interface toolkit that was an add-on to JDK 1.1. Swing was (and is) very powerful, but it is complex, and it gained a reputation, at least partially justified, for being slow and unwieldy.

Swing was integrated into the next major release of the Java platform, which was given the name Java 2 Standard Edition, to distinguish it from the Enterprise Edition, which was focused on the application programming interfaces (APIs) need to build web applications. The other major desktop feature in the Java 2 SE, or JDK 1.2 as it was also known, was Java2D. Java2D greatly improved the graphical capabilities of the platform, with improved font support, the ability to treat text as a shape, the ability to rotate, shear, and scale shapes, and a host of other features. All of this was implemented by a new *graphics pipeline*, which was powerful but also slow. Swing applications originally implemented on JDK 1.1 ran more slowly on the Java 2 platform. In addition to this, the Java2D APIs were seen as hard to learn and difficult to use. Interest in Java as a desktop platform began to wane as more and more companies turned to web applications to satisfy their needs. The performance problems in Java2D and Swing were addressed with subsequent releases so that by the time JDK 1.4 appeared, desktop applications written with Swing could outperform those running on the now obsolete JDK 1.1, but this was not enough to prompt a serious revival in the fortunes of desktop Java.

Fortunately for Java desktop developers, at the JavaOne conference in May 2007, Sun Microsystems made several announcements that were aimed at reclaiming the desktop from other vendors that had moved in to satisfy the need for a rich client-side platform—specifically Adobe with Flash and Flex and Microsoft with its newly announced Silverlight product. Sun announced a major overhaul of its client-side technologies, beginning with J2SE itself and culminating with a new technology called JavaFX.

What Is JavaFX?

Sun's marketing organization describes JavaFX as a new platform for writing *rich Internet applications* (RIAs), or to those of us who would prefer to skip the marketing hype, it is a new language and runtime environment that lets you write rich client applications and then deploy them to users' desktops, mobile devices, Blu-ray players, and TVs (provided that they have the appropriate supporting environment). For a high-level description of the language and its runtime, see the introduction in Chapter 1, "An Overview of JavaFX."

JavaFX runs on the Java platform. The early releases were exclusively targeted at the Microsoft Windows and Apple Mac OS X platforms, but there are now fully supported versions for Linux and OpenSolaris, and a developer release that runs on mobile devices based on Windows Mobile is also available. By the time you read this book, it is likely that you can buy mobile handsets that include a fully supported JavaFX software stack.

From a developer's point of view, writing JavaFX applications couldn't be easier. If you are familiar with Java or a similar language such as C++, you will find the transition to the syntax of the JavaFX Script language easy to make. Much of what you already know remains true in JavaFX, and there are some new features that, once you use them, you will wonder how you ever managed to work without them. The most obvious of these is *binding*, which enables you to link one piece of application state to another.

There is a whole chapter on binding in this book, but there is a simple introduction to the concept in Chapter 1.

The JavaFX user interface libraries offer a clean and very easy-to-use API. If you are a Swing developer, you will recognize many of the concepts, but you may be disappointed at first because the feature set is smaller than that provided by Swing. This is, of course, only a temporary state of affairs. Over time, the API will be expanded, and it should eventually be possible to do in JavaFX almost anything that Swing allows today.[1] On the plus side, the concepts of the *scene graph* and *nodes* make it easy to build impressive user interfaces. As you'll see in the first few chapters of this book, the features built in to the scene graph APIs let you accomplish in a few lines of JavaFX code things that would have taken many more lines of Java code and also an intimate knowledge of the Java2D APIs. Even better, the resulting code will run (with a few exceptions) on both your desktop and on your mobile phone, so you don't need to work with two different APIs and two different development environments to create a truly portable application.[2]

For those of us who are developers first and artists second (or, like me, not an artist at all), JavaFX makes it easy to work with professional graphic designers to create a user interface that doesn't look like it was designed by a programmer, to be used by another programmer. You can import graphic elements or an entire user interface that was originally created in Adobe Illustrator or Adobe Photoshop and then animate it in response to user actions or the passage of time. This capability is provided by the JavaFX Production Suite, a separate download that contains plug-ins that export graphics from the Adobe development tools in a format that can be read by the JavaFX runtime.

If you are reading this book because you expect to use JavaFX at your place of work, you probably already have an audience of users waiting for your application, but if you are intending to develop an application as a private enterprise and you would like to make some money from it, you could try posting it in the *Java Warehouse*. JavaFX applications in the Java Warehouse appear in and can be sold from the *Java Store*. The Java Warehouse and Java Store were announced at JavaOne in 2009 and should represent a revenue opportunity for talented Java and JavaFX programmers. You can find the Java Warehouse at http://java.sun.com/warehouse and the Java Store at http://www.java.sun.com/store.

[1] In the meanwhile, you can always use Java APIs to do almost anything that you can't do directly in JavaFX.

[2] Of course, there are limitations to this. It is easily possible to write a JavaFX application that works well on the desktop but is completely unusable on a mobile device, primarily because mobile devices have smaller screens and less powerful processors. However, if you are careful, it is certainly possible to write a JavaFX application that works on more than one type of device.

How This Book Is Organized

This book is a complete guide to the JavaFX platform—the language, the user interface libraries, and the tools that you can use to develop JavaFX applications. It is divided into three parts. The first part contains an introduction to the platform and a detailed description of the JavaFX script programming language, the second discusses the user interface libraries, and the third covers the APIs that let you access external systems and the tools provided to let you package and deploy your JavaFX applications. Here is an overview of what you'll find in each chapter.

- Chapter 1, "An Overview of JavaFX," contains an overview of the JavaFX platform, the JavaFX script language, and the tools for development and deployment of JavaFX applications.
- Chapter 2, "JavaFX Script Basics," introduces the JavaFX script language by looking at the structure of a JavaFX script file, how scripts are compiled, and how they are executed.
- Chapter 3, "JavaFX Development," shows you how to create a new JavaFX project in both the NetBeans and Eclipse integrated development environments (IDEs) and walks you through the coding, compilation, and execution of a simple JavaFX application. You also see how to compile and run JavaFX applications from the command line and how to extract documentation from JavaFX source files.
- Chapter 4, "A Simple JavaFX Application," builds a more complex JavaFX application and then shows you how to run it as a desktop application, an applet, and on an emulated mobile device. The second half of this chapter shows you how to debug and profile JavaFX code.
- Chapter 5, "Variables and Data Types," is the first of nine chapters that describe the JavaFX script language in detail, beginning with what you need to know about variables and the data types that the language supports. We also discuss the support that the JavaFX runtime provides for the internationalization of applications that need to support more than one native language.
- Chapter 6, "Expressions, Functions, and Object Literals," discusses the arithmetic, Boolean, and other operators that the language provides and introduces the two types of functions that exist in JavaFX. JavaFX functions are first-class citizens of the language, which means, among other things, that you can have a variable that refers to a function, and you can pass a function reference as an argument to another function. This chapter also discusses object literals, which are the nearest thing that JavaFX has to a Java constructor.
- Chapter 7, "Sequences," introduces one of the more powerful features of JavaFX—sequences. Although they are superficially like Java arrays, the ability of sequences to report changes to their content, when used together with either binding or triggers, makes it easy to create user interfaces that display or let the user manipulate lists of objects.

- Chapter 8, "Controlling Program Flow," covers the JavaFX keywords that enable you to change the flow of control in an application. The `if`, `while`, and `for` statements are, as you might expect, similar to their Java equivalents, but there are some important differences. For example, `if` is an expression, which means it can return a value, and `for` operates on a sequence of values and may return a sequence of derived values.
- Chapter 9, "Binding," discusses what is probably the single most important feature of JavaFX. The `bind` keyword enables you to create an association between an expression and a variable so that whenever the value of the expression changes, its new value is written to the variable without programmer intervention. As you'll see in this chapter and throughout the book, this makes it much easier to create user interfaces in JavaFX than it is in Java.
- Chapter 10, "Triggers," introduces the trigger mechanism, which allows arbitrary code to be run when the value of a variable changes.
- Chapter 11, "JavaFX Script Classes," shows you how to write your own JavaFX classes. Unlike Java, code in JavaFX does not have to be coded inside an explicit class definition. As a result, it is possible to write complete applications without knowing how to define a JavaFX class. However, if you want to create a library of reusable code, you need to create your own JavaFX classes.
- Chapter 12, "Platform APIs," covers a subset of the JavaFX runtime that is not part of the user interface libraries, including APIs that allow you to access application parameters and system properties and a couple of classes that allow even untrusted applications to store information on a user's computer.
- Chapter 13, "Reflection," discusses the reflective capabilities of the JavaFX platform. Reflection is a feature that can be of great use to developers who write frameworks or software tools where the data types of the objects being manipulated are not always known in advance. This chapter covers all the JavaFX reflection APIs.
- Chapter 14, "User Interface Basics," is the opening chapter of the part of the book that covers the user interface classes. It introduces the `Stage` and `Scene` classes, which represent the top-level container of an application, and provides a high-level view of the scene graph and the nodes from which it is composed.
- Chapter 15, "Node Variables and Events," takes a detailed look at the variables that are common to all nodes and the node events that your application registers for to track mouse and keyboard activity. It also discusses colors and color gradients.
- Chapter 16, "Shapes, Text, and Images," opens with a discussion of the basic node types that are provided by the JavaFX runtime (such as rectangles, circles, ellipses, and so on) and covers font handling and the rendering of text in JavaFX. Finally, you'll see how to load, display, and, if necessary, resize an image.
- Chapter 17, "Coordinates, Transformations, and Layout," describes the coordinate system used to place and describe the size of a node and the transformations, such

as rotation, translation, and scaling, that can be applied to a node. The second half of the chapter introduces the `Container` class and a variety of its subclasses that have an embedded node layout policy.

- Chapter 18, "Animation," describes the animation features of JavaFX. Using just constructs, you can change the position or appearance of a node or group of nodes over a period of time to give the impression of an animation. This chapter covers both the `Timeline` class, which is the basis for JavaFX animation, and a set of higher-level classes called transitions that let you specify an animation in a more abstract fashion.
- Chapter 19, "Video and Audio," covers the JavaFX classes that let you play video and audio clips. As you'll see, JavaFX supports several platform-specific media formats (including MP3, WMV, and AVI) and a cross-platform format that can be played on any platform that supports JavaFX.
- Chapter 20, "Effects and Blending," shows you how to apply a range of graphical effects to a node. The effects supported include shadowing, blurring, and various different lighting effects. Effects are currently supported only on the desktop platform.
- Chapter 21, "Importing Graphics," describes how you, as a JavaFX developer, can work with a designer to create graphics that you can then import into your application. The JavaFX Production Suite, which is downloaded separately from the JavaFX SDK, provides plug-ins that enable a graphic designer to prepare a graphic in Adobe Photoshop or Adobe Illustrator and export it in a form that is suitable for import into a JavaFX application. You'll see a couple of examples that illustrate the overall workflow, and another that shows you how to capture graphics created in a graphics tool that creates output in SVG (Scalable Vector Graphics) format.
- Chapter 22, "Cross-Platform Controls," discusses the node-based controls that allow user input and display lists of data and other information to the user. These controls work on the desktop, on mobile devices, and with Java TV.
- Chapter 23, "Style Sheets," shows you how to change the appearance of your application without changing a line of code by using a style sheet. The style sheets supported by the JavaFX runtime are similar to the Cascading Style Sheets (CSS) documents used with HTML and provide many of the same features. Style sheets work with the basic node classes, controls, and custom controls.
- Chapter 24, "Using Swing Controls," shows you how to embed Swing components into your JavaFX application. An embedded Swing component is a node, which means that you can rotate it, scale it, shear it, change its opacity, and so on. The JavaFX runtime contains equivalents for many of the standard Swing components, and those that are not directly supported, including third-party components, can be made available to a JavaFX application through a wrapper class.

- Chapter 25, "Building Custom Nodes," describes how to create your own custom nodes, controls, and layouts and how to use style sheets to change their appearance or behavior.
- Chapter 26, "Charts," covers the JavaFX classes that let you render data in the form of a chart. Support is provided for various different types of charts, including pie charts, line charts, area charts, and scatter charts. All these classes are highly customizable, either in code or from a style sheet.
- Chapter 27, "Using External Data Sources," discusses the support in the JavaFX runtime for retrieving data from a web server or a web service and presenting it to the user. At the lowest level, you can use the `HttpRequest` class to make an asynchronous call to a web server. If the data is returned in Extensible Markup Language (XML) or JavaScript Object Notation (JSON) format, you can use the `PullParser` class to parse it and then display the results in your user interface. At a higher level, JavaFX provides a couple of classes that read an RSS or Atom feed and convert the content to JavaFX objects, thus saving you the trouble of parsing it yourself.
- Chapter 28, "Packaging and Deployment," covers the `javafxpackager` command, which allows you to package your JavaFX application and shows you how to deploy it for delivery as a desktop application, an applet, or to a mobile device.

This book is *not* intended to be read from front to back. Most chapters have a mixture of basic and advanced material, so you definitely do not need to read everything in a chapter before you can progress to the next. I recommend that you install the software that you need, get the example source code, and then read the first four chapters. At this point, you should have a good grasp of the fundamentals of the language and how to compile and run JavaFX application.

Given that this book is aimed mainly at developers who have experience with Java, the odds are good that you will be able to get a grip on the JavaFX Script language very quickly, and you should only need to skim over the language chapters before diving into the in-depth coverage of the graphical user interface (GUI) libraries in Part III, " User Interfaces with JavaFX." You'll find that each of the GUI chapters contains detailed information on the topic that it covers, so feel free to skip from chapter to chapter picking out the information that you need from each. When you need more detail on a language feature, you can return to the relevant chapter in Part II, "The JavaFX Script Language," and read about it.

Getting the Software

This book includes more than 400 example JavaFX applications. Most of them are small and are focused on a single aspect of the language or the runtime libraries. To get the best from this book, you need to download and install both the example source code and the JavaFX runtime and documentation, as described in the paragraphs that follow. In addition to the software listed here, you need to have a Java Development Kit installed. The minimum requirement for running JavaFX applications is Java 5, but I recommend that you get the latest release of the platform that is currently available. To use all the features of the JavaFX platform, you need at least Java 6 update 10. You can download the JDK from http://java.sun.com/javase/downloads/index.jsp.

The JavaFX SDK

There are two ways to install a JavaFX runtime on your computer: You can get the JavaFX SDK, which allows you to develop applications using an editor of your choice and a set of command line tools; or you can get the JavaFX plug-in for either the NetBeans or Eclipse IDEs. In fact, it is a good idea to get both the SDK and an IDE plug-in, because the API documentation in the SDK is, at least in my opinion, easier to use than the help facility in an IDE, while the IDEs allow you to find your way around the APIs more easily because they provide code completion, automatic management of imports, and an easy way to execute your applications either on the desktop or on a mobile device emulator.

To download the SDK, go to http://javafx.com. There, you will find a link to the downloads page where you will find most of the additional software that you need. The `bin` directory of the SDK contains command-line tools that let you run the compiler, package an application for deployment, extract and format JavaFX documentation, and run a compiled application. You'll find documentation for the command-lines tools in the `docs` directory and the API documentation for the platform in the directory `docs/api`. It is worth bookmarking the API documentation in your browser because you will probably refer to it frequently.

The text and examples in this book refer to and have been tested with JavaFX version 1.3, which was released in April 2010.

The NetBeans IDE

If you'd like to use the NetBeans IDE to compile and run the example source code for this book, you can get it from the download page at http://javafx.com. Alternatively, you can download it from the NetBeans site at http://www.netbeans.org/features/javafx. Be sure to choose a package that contains the JavaFX plug-in.

If you already have the IDE (version 6.9 or higher supports JavaFX 1.3), you can add the JavaFX plug-in by going to Tools, Plugins, opening the Available Plugins tab, and installing the JavaFX Kit and JavaFX SDK plug-ins. If these plug-ins do not appear, click the Reload Catalog button to update the list.

Having installed the IDE, you need to install the plug-ins for web application development, which are required for the examples in Chapter 27, "Using External Data Sources." See the "GlassFish" section, below, for further information.

As new releases of the JavaFX SDK appear, you can update the IDE plug-in from the Installed tab of the Plugins window. This is the recommended approach because it is simple and ensures that you have the latest IDE features for JavaFX. Alternatively, you can download and install a new SDK and then make it available in the IDE as follows:

1. Go to Tools, Java Platforms, and click Add Platform.
2. Select JavaFX Platform and click Next.
3. Assign a name for the SDK and navigate to its install directory, and then click Finish.

You can assign your new SDK version to a JavaFX project by right-clicking the project node in the Projects window, selecting Properties and the Libraries, and choosing the SDK to be used from the JavaFX Platform combo box.

The Eclipse Plug-In for JavaFX

There are two Eclipse plug-ins for JavaFX, one provided by Sun Microsystems, the other by Exadel (http://www.exadel.com). In this book, we use the Sun Microsystems plug-in, which requires Eclipse version 3.4 or later.

Plug-In Installation and Update for Eclipse 3.4

To install the Sun Microsystems plug-in, do the following:

1. On the Eclipse main menu bar, choose Help, Software Updates, and then open the Available Software tab.
2. Click the Add Site button and enter the following URL into the Add Site dialog: **http://javafx.com/downloads/eclipse-plugin/**.
3. The site will be added to the available software list. Open the site node to show the JavaFX Features node and select it.
4. Click the Install button to install the plug-in.

You can update the plug-in to a later release, or check whether there is an update, as follows:

1. Select Help, Software Updates, and then open the Installed Software tab.
2. Select the row for JavaFX Feature.
3. Click Update.

If an update is available, you will be prompted to accept the license, and then the update will be installed.

Plug-In Installation and Update for Eclipse 3.5

The steps required to install the plug-in for Eclipse 3.5 are as follows:

1. On the main menu bar, select Help, Install New Software.
2. In the Work With field on the Available Software dialog, enter the URL **http://javafx.com/downloads/eclipse-plugin/**, and then click Add.
3. In the Add Site dialog that appears, give the site a name (for example, JavaFX Plugin) and click OK.

4. After a short pause, an entry for JavaFX Features appears in the Available Software dialog. Select the check box for this entry, and then click Next.
5. The plug-in details will be obtained and displayed. Click Next, and on the next page, review the license. If it is acceptable, select I Accept the Terms of the License Agreement, and then click Finish to install the plug-in.

To update the plug-in to a later release, or to check whether an update is available, select Help. Check for Updates and follow the instructions to install a new version of the plug-in.

GlassFish

You need a web server, such as the one that comes with the GlassFish application server, to run the examples for Chapter 27.You also need to install plug-ins for NetBeans or Eclipse that let you work with GlassFish from within the IDE. In this section, we describe how to work with GlassFish from within NetBeans.

Installing GlassFish

You can get GlassFish from the GlassFish community downloads page at https://glassfish.dev.java.net/public/downloadsindex.html. I used GlassFish version 2 when writing this book, but the examples should also work with version 3. During the installation process, you will be asked for the username and password to be used for server administration.You must supply these values again when registering the application server with the NetBeans or Eclipse IDE plug-in.

Installing the NetBeans Plug-Ins

To install the plug-ins required to work with the GlassFish application server from within the NetBeans IDE, do the following:

1. From the main menu, select Tools, Plugins, and open the Available Plugins tab.
2. Select Java Web Applications and click Install. Follow the prompts until the plug-in, and a few other plug-ins that it requires, is installed.
3. You now need to register you GlassFish server with the plug-in, which you can do as follows:
    a. On the main menu, select Window, Services. This opens the Service view.
    b. Right-click the Servers node and select Add Server.
    c. In the Add Server Instance dialog, select the version of GlassFish that you installed and click Next.
    d. In the Server Location field, enter the installation directory of your GlassFish server and click Next.
    e. Enter the administrator username and password that you assigned when installing GlassFish and click Finish.

Your GlassFish server should now appear under the Servers node in the Services view. If you expand the Databases node, you should see a JavaDB entry with URL jdbc:derby://localhost:1527/sample. Start the database server by right-clicking the sample node and selecting Connect. When the database starts, expand the `APP` node and then the `Tables` node, and you should see the `CUSTOMER` table from the sample database. This table will be used in Chapter 27.

The JavaFX Production Suite

The JavaFX Production Suite provides plug-ins for Adobe Photoshop and Adobe Illustrator that let you export artwork in a form that can be easily imported into a JavaFX application. You'll find a full description of the Production Suite, together with installation instructions, in Chapter 21, "Importing Graphics."

JavaDB

One of the examples in Chapter 27 uses the JavaDB database. If you use Windows and you have the Java 6 JDK installed, you already have JavaDB on your computer—you'll find it at `C:\Program Files\Sun\JavaDB`. Otherwise, you should download JavaDB from http://developers.sun.com/javadb/downloads/index.jsp and install it.

The Example Source Code

A Zip file containing the example source code for this book can be found at http://www.informit.com/title/9780321601650. (Click on the Downloads tab.) Extract the content of the file into a directory and then set the value of the `javafxsdk.dir` property in the `build.properties` file so that it points to the directory in which you installed the JavaFX SDK. You can then import the source code into the NetBeans or Eclipse IDE.

> **Note**
>
> You need to set this property only if you plan to build and run the examples from the command line, which requires that you also download the JavaFX SDK. If you plan to use an IDE to build and run the examples, you do not need to edit this file. You will find instructions for building and running the examples from the command line in Appendix A, "Using JavaFX Command-Line Tools."

Importing the Example Source Code into the NetBeans IDE

Assuming that you have installed all the relevant plug-ins, you can import the example source code into NetBeans as follows:

1. On the main menu, select File, Open Project.
2. In the Open Project dialog, navigate to the directory in which you installed the example source code and then into the `desktop` subdirectory.

3. Select the project file in that directory (it's called JavaFX Book Desktop NetBeans Project) and click Open Project.

You may get a message about a missing reference, which we will resolve shortly. Close the message dialog and the project will be imported.

Repeat this process for the projects in the subdirectories `gui`, `moregui`, `intro`, `language`, and `server/ExternalData`.

To fix the missing reference problem, do the following:

1. Right-click the node for the JavaFX Book Desktop NetBeans Project and select Resolve Reference Problems.
2. In the Resolve Reference Problems dialog, click Resolve to open the Library Manager dialog, and then click New Library.
3. In the New Library dialog, enter the name **JavaDBJavaFXBook** and click OK.
4. In the Library Manager dialog, click Add JAR/Folder and navigate to the directory in which JavaDB is installed. If you run the Java 6 JDK on Windows, you will find JavaDB in the folder `C:\Program Files\Sun\JavaDB`. Otherwise, you should install JavaDB as described in the section "JavaDB," earlier in this Preface. Navigate to the `lib` directory and select the file `derbyclient.jar`. Click OK, and then click OK to close the Library Manager.

To run any of the examples, right-click the source file and select Run File. You'll find more information on how to run the examples in Chapter 3, "JavaFX Script Development," and Chapter 4, "A Simple JavaFX Application."

Importing the Example Source Code into the Eclipse IDE

If you have the JavaFX plug-in installed, you can build and run the example source code in Eclipse.

> **Warning**
>
> Compiling all the example source code requires a lot of Java heap space. To make sure that you don't run out of memory, specify a larger heap when running Eclipse, like this:
>
> `eclipse.exe -vmargs -Xmx1024M`

To import the example source code, do the following:

1. On the main menu, select File, Import.
2. In the Import dialog, select General, Existing Projects into Workspace, and then click Next.
3. Enter the directory in which you installed the example source code as the root directory.
4. Select all the projects that appear in the Projects list, and then click Finish.

To run an example, right-click the JavaFX file and select Run As, JavaFX Application. When the Edit Configuration dialog appears, click Run. You'll find more information on how to run the examples in Chapters 3 and 4.

Conventions

`Courier` font is used to indicate JavaFX and Java code, both in code listings and in the code extracts embedded in the text. Lines of code that are of particular interest are highlighted in bold.

Throughout this book, you will find tables that list the accessible variables of JavaFX classes. Each row corresponds to one variable, and the columns contain the variable's name, its type, the permitted modes of access, its default value, and a description. Here's an example:

Variable	Type	Access	Default	Description
focused	Boolean	R	(None)	Whether the Stage is focused
icons	Image[]	RW	Empty	The icons used in the title bar of the top-level container
title	String	RW	Empty string	The title used in the title bar of the top-level container
visible	Boolean	RW	true	The visibility of the Stage

The value in the Access column contains the permitted access modes for application code (more specifically, for code that is not related to the owning class—that is, code that is not in the same package as the class or in a subclass). The possible access modes are as follows.

R	The value can be read.
W	The value can be written at any time.
I	The value can be set, but only when an instance of the class is being created (that is, at initialization time).

Further Information

Even though JavaFX is a recent innovation, there are already many sources that you can refer to for up-to-date information. The most obvious of these is the JavaFX website at http://javafx.com, where you can download the latest release, find hints and tips, and browse through a gallery of examples. There is also a set of forums dedicated to JavaFX at http://forums.sun.com/category.jspa?categoryID=132 and a JavaFX blog at http://blogs.sun.com/javafx/.

JavaFX is still a young technology and currently lacks some of the features that you'll find in Swing or in comparable toolkits. If you can't find what you need in the JavaFX runtime, you might find it instead at http://jfxtras.org/, a site dedicated to the development, discussion, and extension of the JavaFX platform. Here you will find third-party controls, shapes, layout containers, and lots of sample code to help you get the most out of the platform.

Feedback

Although this book has been reviewed for technical accuracy, it is inevitable that some errors remain. If you find something that you think needs to be corrected or that could be improved, or if there is something that you think could usefully be added in future editions of this book, please contact the author by e-mail at kimtopley@gmail.com.

Acknowledgments

This is the eighth time that I have been through the process of writing a book. Two of those books didn't see the light of day, but the six that did (one of them being a second edition) have been interesting, character-building experiences. There is no doubt that writing a book involves a major commitment of time and effort. This book was conceived when the first version of JavaFX was officially announced at JavaOne in 2007. By the time I finished the first draft, I had been working on it in my spare time for more than two years. Naturally, I blame the time overrun on Sun Microsystems, who caused me to discard at least 300 pages of text by throwing away virtually all the language and libraries between the announcement in 2007 and the next JavaOne in May 2008, but in so doing creating a much better product.

I am, once more, indebted to my editor, Greg Doench, who accepted my proposal for this book based on a very sketchy outline and shepherded the whole project to completion despite my attempts to frustrate the process by repeatedly adding "just one more" section to an ever-growing table of contents. Thanks also to Michael Thurston, Keith Cline, Julie Nahil, and the rest of the Addison-Wesley team who converted this book from a raw manuscript to the final result that you now hold in your hands. Along the way, reviewers Peter Pilgrim and Joe Bowbeer provided insightful technical feedback for which I'm very grateful.

I was lucky enough to receive help from members of the JavaFX team at Sun Microsystems while writing this book. My thanks go, in no particular order, to Brian Goetz, Amy Fowler, Richard Bair, Robert Field, and Per Bothner, who took the time to answer my questions even when they were busy trying to meet deadlines of their own.

20

Effects and Blending

In this chapter, you see how to use the classes in the `javafx.scene.effects` and `javafx.scene.effects.lighting` packages, which implement graphical effects that you can use to enhance the appearance of your application. After discussing effects in general, the first part of this chapter describes 15 different effects that you can use to create blurs, shadows, warps, and various lighting effects. The second part describes the blending effect, which provides 19 different ways to combine two inputs, such as a node and another effect, to produce an output. The same 19 blending modes can also be applied to a group (and therefore also to a container) to control how the pixels for intersecting nodes are combined. The last part of this chapter looks at the ways in which you can light a scene by using the `Lighting` effect.

Effects are a feature of the desktop profile—they do not work on mobile devices—so the example source code for this chapter can all be found in the `javafxeffects` package in the JavaFX Book Desktop project. You can use the conditional feature API described in Chapter 12, "Platform API," to determine at runtime whether effects are available to your application.

Effects Overview

An effect is a graphical filter that accepts an input (and in some cases more than one input) and modifies it in some way to produce an output. The output is either rendered as part of a scene or becomes the input for another effect. The combination of a node and one or more effects is referred to here as an *effects chain*.

Effects Chains

Figure 20-1 shows two common effects chains. An effects chain contains, at minimum, one node and one effect.

In the first chain, at the top of the figure, a single effect is applied to a node, and the result is drawn onto the scene. The second chain contains two effects. In this case, the first effect is applied to the node, which results in an output image that becomes the input for the second effect. It is the output of the second effect that will be drawn onto the scene.

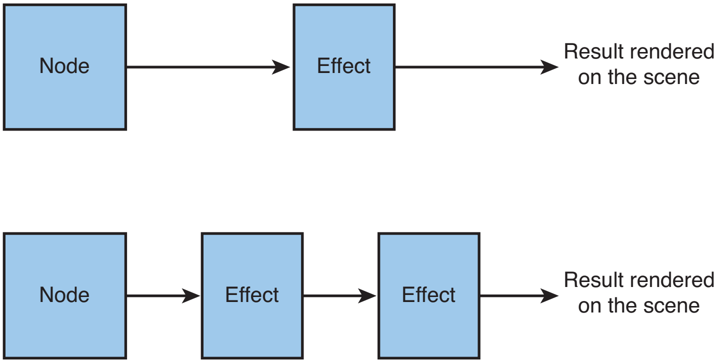

Figure 20-1 Relationship between effects and nodes

Effects and Nodes

When an effect is applied to a node, the output of the effects chain logically replaces the node itself on the screen. In general, an effect will change the bounds of a node. For example, adding a shadow to a node by using the `DropShadow` effect will typically make it wider and taller. The node's local and parent bounds are adjusted based on the result of the effects chain, but its layout bounds are not affected. When a node has both effects and transformations, the effect is applied before the transformations. This means, for example, that adding a shadow and then scaling up the node will also scale up the shadow.

An effect is linked to a node via its `effect` variable.[1] The code in Listing 20-1, which you will find in the file `javafxeffects/Effects1.fx`, shows how simple it is to add an effect to a node. In this case, a drop shadow is added by the three lines of code starting on line 19. Figure 20-2 shows the result.

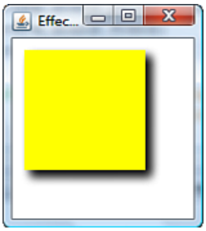

Figure 20-2 A rectangle with a drop shadow effect

[1] This variable does not exist in the mobile profile. If you get compilation errors when trying to run the examples for this chapter, the reason is most likely that you are trying to build for the mobile emulator.

Listing 20-1 **Adding an Effect to a Node**

```
package javafxeffects;

import javafx.scene.effect.DropShadow;
import javafx.scene.paint.Color;
import javafx.scene.Scene;
import javafx.scene.shape.Rectangle;
import javafx.stage.Stage;

var rect: Rectangle;
Stage {
    title: "Effects #1"
    scene: Scene {
        width: 150 height: 150
        content: [
            rect = Rectangle {
                x: 10 y: 10
                width: 100 height: 100
                fill: Color.YELLOW
                effect: DropShadow {
                    offsetX: 5 offsetY: 5
                }
            }
        ]
    }
}
println("Layout bounds: {rect.layoutBounds}");
println("Parent bounds: {rect.boundsInParent}");
```

The last two lines of Listing 20-1 print the layout bounds and parent bounds of the rectangle. Here's the result:

```
Layout bounds: BoundingBox [minX = 10.0, minY=10.0, maxX=110.0, maxY=110.0,
width=100.0, height=100.0]
Parent bounds: BoundingBox [minX = 6.0, minY=6.0, maxX=124.0,
maxY=124.0,width=118.0, height=118.0]
```

As you can see, the rectangle's layout bounds correspond to its specified width and height (because the layout bounds do not include the results of the effect), but width and height of the parent bounds have both increased from `100` to `118` because of the space occupied by the drop shadow.

Applying more than one effect is simply a matter of linking one effect to another. The following code (which you'll find in the file `javafxeffects/Effects2.fx`) adds a reflection to the drop shadow, giving the result shown in Figure 20-3:

```
Rectangle {
    x: 10
    y: 10
```

```
    width: 100
    height: 100
    fill: Color.YELLOW
    effect: Reflection {
        input: DropShadow {
            offsetX: 5 offsetY: 5
        }
    }
}
```

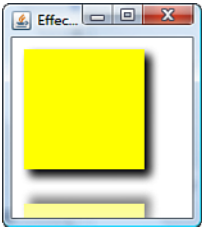

Figure 20-3 Applying two effects to the same rectangle

The linkage between the effects is made through the `input` variable of the reflection effect—the drop shadow is applied first, and the result of this becomes the input to the reflection effect. When no input is specified, the node itself is used as the input, as in the case of the drop shadow effect.

Not all effects have an `input` variable. Those that don't can only appear as the first (or only) entry in the effects chain. The `DropShadow` class is an example of this.[2] Other effects can have more than one input, such as the `Blend` effect that you'll see in the second part of this chapter.

As noted earlier, transformations are applied after any effects, so they apply to the effects, too. The following code, from the file `javafxeffects/Effects3.fx`, adds a rotation to the two effects that are applied to the `Rectangle`, as shown in Figure 20-4.

```
Rectangle {
    x: 10 y: 10
    width: 100 height: 100
    fill: Color.YELLOW
    rotate: -45
    effect: Reflection {
        input: DropShadow {
```

[2] The lack of an `input` variable in the `DropShadow` and other effects classes may be a temporary state of affairs. An issue has been filed at http://javafx-jira.kenai.com that may result in this being changed.

```
        offsetX: 5 offsetY: 5
    }
}
```

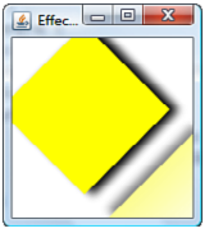

Figure 20-4 Using effects and transformations together.

Effects and Groups

A particularly powerful feature of effects is that they can be applied to a group. An effect that is applied to a group operates on the group as a whole. This is particularly useful if you want to create an effect that is uniform across the scene, such as the direction of lighting.

The following code, from the file `javafxeffects/Effects4.fx`, applies a `DropShadow` effect to a group that contains a rectangle and a circle; as you can see in Figure 20-5, this gives both of the nodes a `DropShadow` effect:

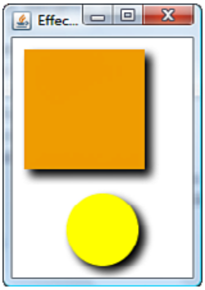

Figure 20-5 Applying an effect to a **`Group`**

```
Group {
    effect: DropShadow {
        offsetX: 5 offsetY: 5
    }
    content: [
        Rectangle {
            x: 10 y: 10
            width: 100 height: 100
```

```
            fill: Color.ORANGE
        }
        Circle {
           centerX: 75 centerY: 160 radius: 30
           fill: Color.YELLOW
        }
   ]
}
```

The JavaFX Effects Classes

The JavaFX SDK provides 17 different effects that can be applied to any node. This section describes and illustrates all the effects, with the exception of `Blend` and `Lighting`, which have sections of their own at the end of the chapter. Each effect has a set of variables that you can use to customize it. As we examine each effect, we'll take a look at the variables available and roughly consider what each of them does. There are too many combinations to illustrate them all in this chapter, so in most cases we limit ourselves to some typical examples. It is easy to experiment with these effects—all you need to do is modify the example source code. You can also use the Effects Playground application that you'll find among the samples at http://javafx.com.

GaussianBlur

The `GaussianBlur` effect produces a blurred version of its input. The "Gaussian" part of the name refers to the fact that the output pixels are calculated by applying a Gaussian function to the source pixel and a group of pixels surrounding it. If you are interested in the details, you'll find them at http://en.wikipedia.org/wiki/Gaussian_blur. The size of the group of adjacent pixels that are used to calculate the result is controlled by the `radius` variable (see Table 20-1). The larger the value of the `radius` variable, the greater the blur effect will be. When the value of this variable is `0`, there is no blur at all.

Table 20-1 **Variables of the `GaussianBlur` Class**

Variable	Type	Access	Default	Description
`input`	`Effect`	RW	`null`	The input to this effect
`radius`	`Number`	RW	`10.0`	The radius of the area containing the source pixels used to create each target pixel, in the range `0` to `63`, inclusive

Two example of the `GaussianBlur` effect applied to the image used in the previous two sections are shown in Figure 20-6. Here's the code used to create this effect, which you'll find in the file `javafxeffects/GaussianBlur1.fx`:

```
ImageView {
    image: Image { url: "{__DIR__}image1.jpg" }
    effect: GaussianBlur {
        radius: bind radiusSlider.value
    }
}
```

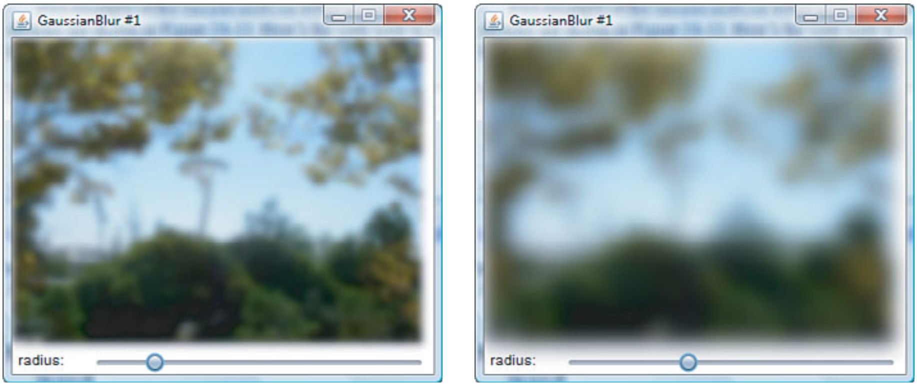

Figure 20-6 The **`GaussianBlur`** effect

The image on the left has a blur radius of 10, while the one on the right has radius 40.

`BoxBlur`

`GaussianBlur` is a high-quality effect, but it is also a relatively expensive one. The `BoxBlur` effect is a cheaper way to produce a blur, albeit one of lower quality. The variables that you can use to control the `BoxBlur` effect are listed in Table 20-2.

Table 20-2 **Variables of the `BoxBlur` Class**

Variable	Type	Access	Default	Description
`input`	`Effect`	RW	`null`	The input to this effect.
`height`	`Number`	RW	`5.0`	The vertical size of the box used to create the blur, in the range `0` to `255`, inclusive.
`width`	`Number`	RW	`5.0`	The horizontal size of the box used to create the blur, in the range `0` to `255`, inclusive.
`iterations`	`Number`	RW	`1`	The number of averaging iterations, in the range `0` to `3`, inclusive. Higher values produce a smoother blur effect.

This effect works by replacing each pixel of the input by the result of averaging its value with those of its neighboring pixels. The pixels that take part in the operation are those in a rectangular area surrounding the source pixel, the dimensions of which are given by the `width` and `height` variables. You can see the effects of changing these variables by running the code in the file `javafxeffects/BoxBlur1.fx`. This example applies the `BoxBlur` effect to the same image as we used to illustrate `GaussianBlur`. The `width`, `height`, and `iterations` variables are set from three sliders that allow you to test the full ranges of values for each variable. Here's how the `BoxBlur` is applied:

```
ImageView {
    image: Image { url: "{__DIR__}image1.jpg" }
    effect: BoxBlur {
        height: bind heightSlider.value
        width: bind widthSlider.value
        iterations: bind iterationSlider.value
    }
}
```

Increasing the value of the `height` variable produces a vertical blur, as shown on the left of Figure 20-7. Similarly, the `width` variable controls the extent of the blur in the horizontal direction.

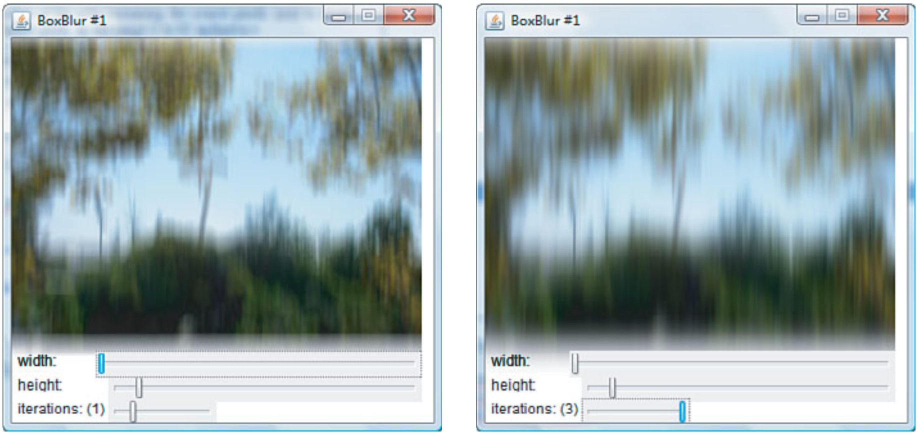

Figure 20-7 The Box Blur effect

You can use the `iterations` variable to increase the quality of the blur at the expense of greater CPU utilization. When this variable has the value `2` or `3`, the averaging operation is repeated the specified number of times. On the second iteration, the averaged pixels are averaged against each other, which tends to smooth out any sharp differences that might exist near to edges in the input source. A third iteration produces an even smoother result. You can see the result of applying three iterations to a horizontal blur of the input image on the right of Figure 20-7. A `BoxBlur` with three iterations produces a result that is close to that of a `GaussianBlur`, but at a slightly lower cost.

MotionBlur

`MotionBlur` creates the effect that you would see if you look out of the window of a fast-moving vehicle. Like `GaussianBlur`, it has a `radius` variable that determines how much of a blur is to be applied. It also has an `angle` variable that lets you specify the direction of the motion. These variables are described in Table 20-3.

Table 20-3 **Variables of the MotionBlur Class**

Variable	Type	Access	Default	Description
`input`	`Effect`	RW	null	The input to this effect
`angle`	`Number`	RW	0	The angle of the motion blur
`radius`	`Number`	RW	10.0	The radius of the area containing the source pixels used to create each target pixel, in the range 0 to 63 inclusive

There are no restrictions on the value of the `angle` variable, but values greater than 360 are treated modulo 360, while negative values are first reduced modulo 360 and then have 180 added to them, so that -90 is the same as 270. The following extract shows how to apply a `MotionBlur` to a node.

```
    image: Image { url: "{__DIR__}image1.jpg" }
    effect: MotionBlur {
        angle: bind angleSlider.value
        radius: bind radiusSlider.value
    }
}
```

If you run the code in the file `javafxeffects/MotionBlur1.fx`, you can experiment with the effects of different `radius` and `angle` values. Two examples with different angles are shown in Figure 20-8. The angle slider lets you vary the value of this variable from –180 when the thumb is at the far left to +180 at the far right. In the example on the left of the figure, the `angle` variable is `0`, which gives a horizontal blur. In the example on the right, the `angle` variable has the value `90`, and the result is a vertical blur. As is the case elsewhere in the JavaFX API, angles are measured with 0 at the 3 o'clock position and increase as you move in a clockwise direction.

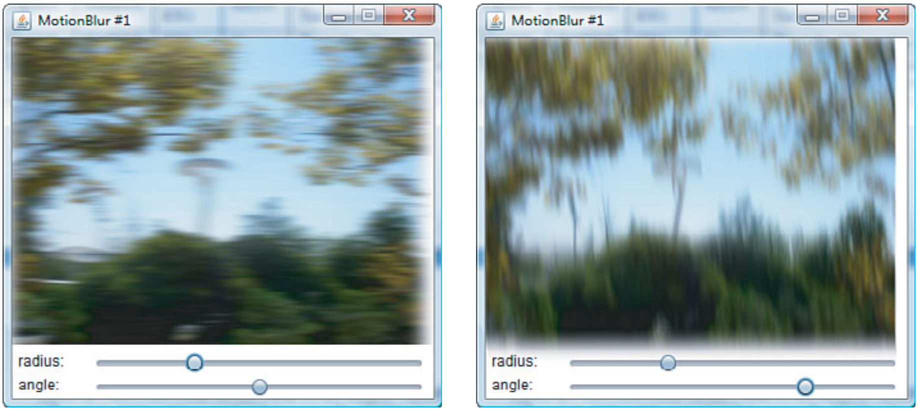

Figure 20-8 The **MotionBlur** effect

DropShadow

As you have already seen, the `DropShadow` effect draws a shadow that appears to be behind and partly obscured by the node to which it is applied. By using this effect with the appropriate variable settings, you can give the impression that the node is floating above a nearby surface or one slightly farther away. You can also change the nature of the shadow to indicate whether the light source is close to or a long way from the node. The variables that you can use to configure the `DropShadow` class are listed in Table 20-4.

Table 20-4 **Variables of the DropShadow Class**

Variable	Type	Access	Default	Description
`blurType`	`BlurType`	RW	`THREE_PASS_BOX`	The type of blur to be used
`color`	`Color`	RW	`Color.BLACK`	The color to be used for the shadow
`offsetX`	`Number`	RW	`0.0`	The x-offset of the shadow
`offsetY`	`Number`	RW	`0.0`	The y-offset of the shadow
`radius`	`Number`	RW	`10`	The radius of the blur effect if a `GaussionBlur` is used
`width`	`Number`	RW	`21`	The width of the blur if `BoxBlur` is used
`height`	`Number`	RW	`21`	The height of the blur if `BoxBlur` is used
`spread`	`Number`	RW	`0.0`	The proportion of the radius (or box for `BoxBlur`) over which the shadow is fully opaque (see text)

The variables that you will most commonly set are `color`, `offsetX`, and `offsetY`. The `color` variable simply determines the color of the solid part of the shadow, which will generally be slightly darker than the background behind the node. By default, the shadow is black. The `offsetX` and `offsetY` variables control the displacement of the shadow relative to the node itself.

The `blurType` variable controls which of the supported types of blur is used at the edges of the shadow. This variable is of type `javafx.scene.effects.BlurType`, which has the following possible values:

- `BlurType.GAUSSIAN`: A `GaussianBlur`
- `BlurType.ONE_PASS_BOX`: A `BoxBlur` with one iteration
- `BlurType.TWO_PASS_BOX`: A `BoxBlur` with two iterations
- `BlurType.THREE_PASS_BOX`: A `BoxBlur` with three iteration

The code in the file `javafxeffects/DropShadow1.fx` creates a scene containing a rectangle with a `DropShadow` effect and a `GaussianBlur`. There are four sliders that let you control some of the variables listed in Table 20-5. You can use this program to experiment with various settings to see how they work. Figure 20-9 shows a typical example.

Table 20-5 **Variables of the `Shadow` Class**

Variable	Type	Access	Default	Description
`blurType`	`BlurType`	RW	`THREE_PASS_BOX`	The type of blur to be used
`input`	`Effect`	RW	`null`	The input to this effect
`color`	`Color`	RW	`Color.BLACK`	The color to be used for the shadow
`radius`	`Number`	RW	`10.0`	The radius of the blur effect if a `GaussianBlur` is used
`width`	`Number`	RW	`21`	The width of the blur if a `BoxBlur` is used
`height`	`Number`	RW	`21`	The height if the blur if a `BoxBlur` is used

The size of the shadow is determined by the values of the `offsetX`, `offsetX`, and `radius` variables. When the `radius` is `0`, the shadow has a sharp edge as shown on the left of Figure 20-10. In this case, the `offsetX` and `offsetY` values are both `15`, so the shadow is offset by 15 pixels to the right of and below the top-left corner of the node, which gives the impression of a light source that is to the left of and above the top of the node. Negative values for the `offsetX` variable would be used for a light source to the right of the node, and negative `offsetY` values for a light source that is below the node.

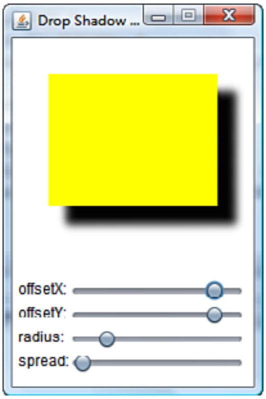

Figure 20-9 Configuring a **DropShadow** effect

Figure 20-10 Effects of the **offsetX**, **offsetY**, and radius variable of the **DropShadow** effect

When the `radius` value is non-0, the edge of the shadow is blurred by blending pixels of the shadow color with those of the background color. The `radius` determines the size of this blurred area. Increasing the `radius` value makes the blurred region, and the size of the shadow, larger, as shown on the right of Figure 20-10. As you can see, the blurring fades out with increasing distance from the original shadow area. The `radius` value can be anywhere between `0` and `63`, inclusive.

By default, the blurred area starts with the shadow color on its inside edge and progresses to the background color on its outside edge. If you want, you can arrange for a larger part of the blurred area to have the shadow color, resulting in a larger, darker shadow. You do this by setting the `spread` value, which ranges from `0.0`, the default, to `1.0`. This value represents the proportion of the blurred area into which the shadow color creeps. On the right side of Figure 20-10, the spread variable has value `0`, and you can see that the shadow gets lighter very rapidly as you move your eyes away from the edge of the rectangle. On the left side of Figure 20-11, the `spread` variable has been set to `0.5`. Now you can see that the darker region of the shadow has increased in size as it encroaches into the blurred area. On the right of Figure 20-11, the `spread` is at of `0.9`, and you can see that almost all the blurred area has been taken over by the shadow color.

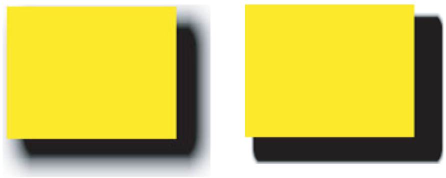

Figure 20-11 The effect of the spread variable

The idea of the `spread` variable is to allow a proper emulation of what would happen if you moved a light source quite close up to the node. A light source nearby would cause a wide shadow, corresponding to a larger blurred area, but it would also cause the darker part of the shadow to increase in size. You simulate the former effect by increasing the blur `radius` and the latter by increasing the `spread`.

InnerShadow

`InnerShadow` is very similar to `DropShadow`, the difference being that the shadow is *inside* the boundaries of the node to which it is applied, rather than outside. This gives the impression of depth within the node, because it appears to have built-up sides. The variables of this class, which are listed in Table 20-6, are almost the same as those of `DropShadow`.

You can see an example of this effect in Figure 20-12. This screenshot shows the result of running the code in the file `javafxeffects/InnerShadow1.fx`. As with the `DropShadow` examples, you can use the sliders to vary the effect parameters and see the results. The `choke` variable is equivalent to the `spread` variable of the `DropShadow` class.

Table 20-6 **Variables of the `InnerShadow` Class**

Variable	Type	Access	Default	Description
`blurType`	`BlurType`	RW	`THREE_PASS_BOX`	The type of blur to be used
`color`	`Color`	RW	`Color.BLACK`	The color to be used for the shadow
`offsetX`	`Number`	RW	`0.0`	The x-offset of the shadow
`offsetY`	`Number`	RW	`0.0`	The y-offset of the shadow
`radius`	`Number`	RW	`10`	The radius of the blur effect if a `GaussianBlur` is used
`width`	`Number`	RW	`21`	The width of the blur if a `BoxBlur` is used
`height`	`Number`	RW	`21`	The height of the blur if a `BoxBlur` is used

Table 20-6 Variables of the `InnerShadow` Class (*Continued*)

Variable	Type	Access	Default	Description
`choke`	`Number`	RW	`0.0`	The proportion of the radius (or box for `BoxBlur`) over which the shadow is fully opaque (see text)

`Shadow`

The `Shadow` effect produces a single-colored and blurred shadow from the node or input effect on which it operates. The extent of the blur depends on the value of the

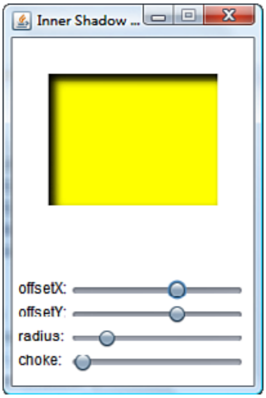

Figure 20-12 Configuring an **`InnerShadow`** effect

`radius`, which is one of the three variables that control this effect, all of which are listed in Table 20-5 on page 661.

You can see an example of this effect as applied to some text in Figure 20-13. You can experiment with different `radius` values by running this example, which you'll find in the file `javafxeffects/Shadow1.fx`:

```
Text {
    textOrigin: TextOrigin.TOP
    x: 30 y: 30
    content: "JavaFX Shadow Effect"
    font: Font { size: 24 }
    effect: Shadow {
        color: Color.BLUE
```

```
        radius: bind radiusSlider.value
    }
}
```

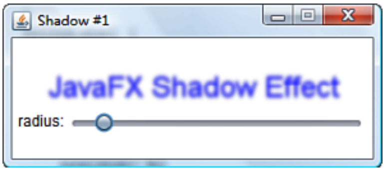

Figure 20-13 The **Shadow** effect

Unlike the other two shadow effects, this one replaces its input instead of augmenting it, so the original text node is not drawn.

Bloom

The `Bloom` effect adds a glow to those areas of its input that are made up of pixels for which the luminosity value is above a given threshold. This effect has only two controlling variable, which are listed in Table 20-7.

Table 20-7 **Variables of the Bloom Class**

Variable	Type	Access	Default	Description
`input`	`Effect`	RW	null	The input to this effect.
`threshold`	`Number`	RW	0.3	The luminosity above which the glow effect will be applied, from 0.0 (all pixels will glow) to 1.0. (No pixels will glow.)

The luminosity of a pixel is a measure of how bright it seems to the human eye. You can see an example of this effect in Figure 20-14, which shows the `Bloom` effect applied to an `ImageView` node[3]:

```
ImageView {
    image: Image { url: "{__DIR__}image1.jpg" }
    effect: bloom = Bloom {
```

[3] You'll find this code in the file `javafxeffects/Bloom1.fx`.

```
            threshold: bind (thresholdSlider.value as Number) / 10
        }
}
```

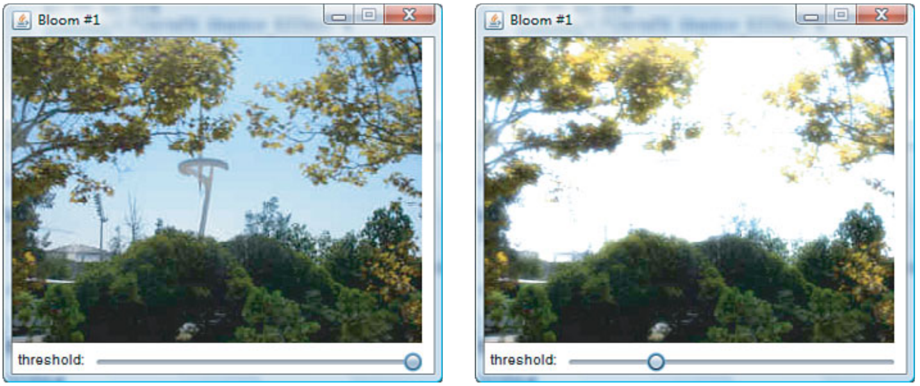

Figure 20-14 The **Bloom** effect

In the image on the left, the threshold value is `1.0`. Because no pixel has a luminosity that is greater than `1.0`, what you see here is the original image. On the right of the figure, the slider has been moved so that the threshold is now set to `0.3`. The blue regions of the image, in particular the sky, are now noticeably brighter. Notice that this effect spills over onto adjacent pixels so that the leaves on the trees near the top of the image have also been brightened.

Glow

`Glow` is very similar to `Bloom`, except that the controlling parameter works in the reverse order. The glow effect makes bright pixels appear brighter. The more of the effect that you apply, as determined by the value of the `level` variable, the brighter those pixels appear. The two variables that control this effect are listed in Table 20-8.

Table 20-8 **Variables of the `Glow` Effect**

Variable	Type	Access	Default	Description
`input`	`Effect`	RW	`null`	The input to this effect.
`level`	`Number`	RW	`0.3`	Controls the intensity of the glow effect. `0.0` gives no glow; `1.0` gives maximum glow.

You'll find an example that allows you to vary the `level` parameter in the file `javafxeffects/Glow1.fx`. The following extract from that file shows how the glow effect is applied to a node:

```
ImageView {
    image: Image { url: "{__DIR__}image1.jpg" }
    effect: Glow {
        level: bind (levelSlider.value as Number) / 10
    }
}
```

Figure 20-15 shows this effect applied to the same image as that used in our discussion of bloom in the previous section. In the image on the left of the figure, the level variable is `0`, so no glow is applied. In the image on the right, the level is set to `0.6`, and the result is almost exactly the same as the result of applying a small amount of bloom to the image, which you can see at the bottom of Figure 20-14. To apply more glow in this example, you move the slider farther to the right, whereas to apply more bloom in the example in the previous section, you moved it farther to the left.

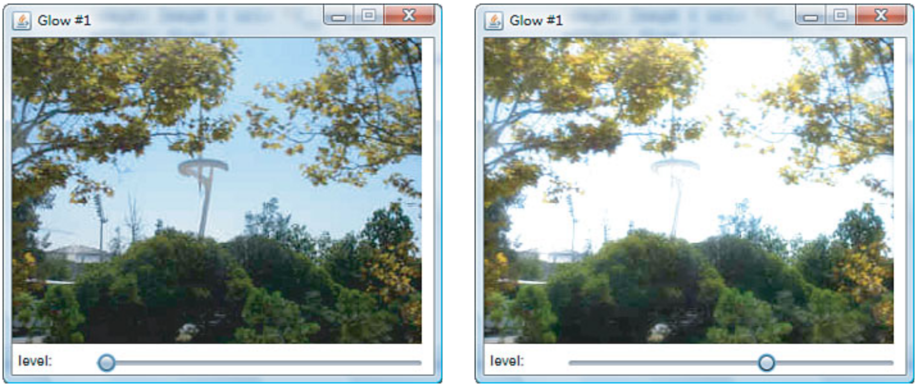

Figure 20-15 The **`Glow`** effect

`Identity`

The `Identity` effect is a little different from the effects that you have seen so far—its sole purpose is to allow an `Image` object to be used as the input to another effect. It is always linked to a node, but that node does not appear in the scene; the result of applying one or more effects to the source image is seen instead. Table 20-9 lists the variables that control the behavior of this class.

The simplest way to explain how these variables work is by using an example. The following code, which you'll find in the file `javafxeffects/Identity1.fx`, applies a `GaussianBlur` effect to an image and places it in the `Scene`.

Table 20-9 **Variables of the `Identity` Class**

Variable	Type	Access	Default	Description
`source`	`Image`	RW	`null`	The source `Image`
`x`	`Number`	RW	`0`	The `x` coordinate of the `Image` relative to the source `Node`
`y`	`Number`	RW	`0`	The `y` coordinate of the `Image` relative to the source `Node`

```
Stage {
    title: "Identity #1"
    scene: Scene {
        width: 380
        height: 280
        var image = Image { url: "{__DIR__}image1.jpg" }
        content: [
            Circle {
                centerX: 100 centerY: 100
                effect: GaussianBlur {
                    input: Identity {
                        source: image
                        x: 10 y: 10
                    }
                    radius: 10
                }
            }
        ]
    }
}
```

The result of running this code is shown in Figure 20-16.

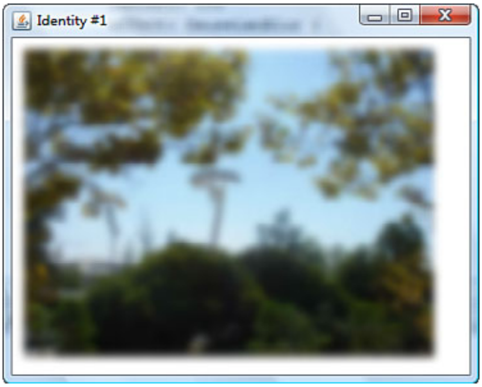

Figure 20-16 The **`Identity`** effect

The `Identity` effect on lines 11 to 14 converts the image to an `Effect` that is then used as the input to the `GaussianBlur`, resulting in a blurred version of the image. These two effects are both linked with a circle, but because the circle is not an input to either of the effects, it does not influence the output, and the blurred image appears instead of it. The only property of the circle that *is* inherited is its coordinate system, which, in this case, is the same as the coordinate system of the scene. The `x` and `y` variables of the `Identity` effect, which are set on line 13, determine where its output would be drawn relative to the circle's coordinate system. In this case, these values cause the image to be placed a little to the right of and below the coordinate origin.

The result of an `Identity` effect, like that of the `Flood` effect that is described in the next section, is often used as one of the inputs to a `Blend` effect, which is discussed later in this chapter.

`Flood`

Like `Identity`, the purpose of the `Flood` effect is to create an input to another effect, in this case a rectangular area filled with a `Paint` or a solid color. The variables that determine the fill color and the bounds of the filled area are listed in Table 20-10.

Table 20-10 **Variables of the `Flood` Class**

Variable	Type	Access	Default	Description
`paint`	`Paint`	RW	`Color.RED`	The `Paint` used to flood the area
`x`	`Number`	RW	`0`	The `x` coordinate of the filled area relative to the source `Node`
`y`	`Number`	RW	`0`	The `y` coordinate of the filled area relative to the source `Node`
`width`	`Number`	RW	`0`	The width of the area to be filled
`height`	`Number`	RW	`0`	The height of the area to be filled

The coordinates and lengths are specified in the coordinate system of the node that this effect is linked with. As with `Identity`, the node itself is replaced in the scene by the result of the effect. The code in the file `javafxeffects/Flood1.fx` uses the `Flood` effect to fill an area with a solid blue color and then applies a `MotionBlur`, giving the result shown in Figure 20-17.

`ColorAdjust`

The `ColorAdjust` effect produces an output that is the result of adjusting some or all the hue, saturation, brightness, and contrast values of its input. The input may be either

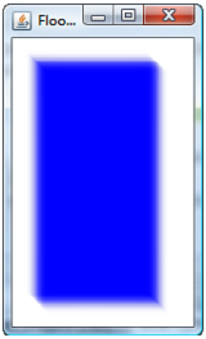

Figure 20-17 The **`Flood`** effect

another effect or a node of any kind, but most commonly an image in an `ImageView` object. The variables of this class are listed in Table 20-11.

You can experiment with this effect by running the example in the file `javafxeffects/ColorAdjust.fx`, which binds a slider to each of the hue, saturation, brightness, and contrast variables of a `ColorAdjust` object that is associated with an `ImageView` node. The values of the hue, saturation, and brightness sliders range from `-1.0` on the left to `1.0` on the right, while the contrast slider provides the value `0.25` in its

Table 20-11 **Variables of the `ColorAdjust` Class**

Variable	Type	Access	Default	Description
`input`	`Effect`	RW	`null`	The input to this effect.
`hue`	`Number`	RW	`0.0`	The amount by which the hue of each pixel should be adjusted, in the range `-1.0` to `1.0`. Value `0` leaves the hue unchanged.
`saturation`	`Number`	RW	`0.0`	The amount by which the saturation of each pixel should be adjusted, in the range `-1.0` to `1.0`. Value `0` leaves the saturation unchanged.
`brightness`	`Number`	RW	`0.0`	The amount by which the brightness of each pixel should be adjusted, in the range `-1.0` to `1.0`. Value `0` leaves the brightness unchanged.
`contrast`	`Number`	RW	`1.0`	The amount by which the contrast should be adjusted, in the range `0.25` to `4`. Value `1` leaves the contrast unchanged.

minimum position and `4.0` at its maximum position. On the left of Figure 20-18, you can see the result of applying almost the maximum brightness, and on the right you see the result of applying the maximum contrast.

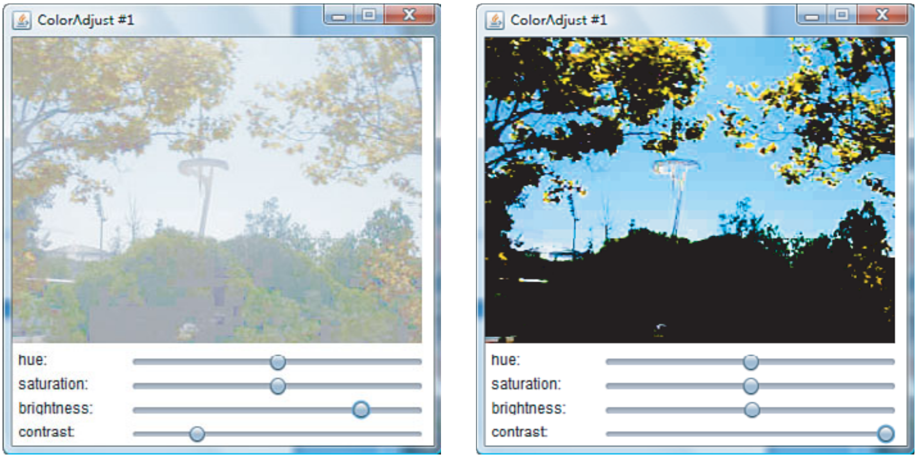

Figure 20-18 The **`ColorAdjust`** effect

`InvertMask`

The `InvertMask` effect takes another `Effect` as its input and produces a result in which all the transparent pixels from the input are opaque and all the opaque pixels are transparent. The output is typically used as one of the inputs to a `Blend` effect, which is discussed later in this chapter. The variables of the `InvertMask` class are listed in Table 20-12.

Table 20-12 **The Variables of the `InvertMask` Class**

Variable	Type	Access	Default	Description
`input`	`Effect`	RW	`null`	The input to this effect
`pad`	`Number`	RW	`0`	The padding to add to the sides of the resulting image

`Reflection`

The `Reflection` effect provides an easy way to create a reflection of a node or group of nodes. The variables that you can use to specify the required characteristics of the reflection are listed in Table 20-13.

Table 20-13 **The Variables of the `Reflection` Class**

Variable	Type	Access	Default	Description
`input`	`Effect`	RW	`null`	The input to this effect
`topOffset`	`Number`	RW	`0`	The distance between the bottom of the input and the beginning of the reflection
`fraction`	`Number`	RW	`0`	The fraction of the input that is used to create the reflection
`topOpacity`	`Number`	RW	`0.5`	The opacity used for the topmost row of pixels in the reflection
`bottom-Opacity`	`Number`	RW	`0`	The opacity of the bottom row of pixels in the reflection

The example code in the file `javafxeffects/Reflection1.fx` allows you to experiment with different values of these variables. A typical result, which is equivalent to the following code, is shown in Figure 20-19.

```
Text {
    content: "JavaFX Developer's Guide"
    x: 20 y: 50
    fill: Color.WHITE
    font: Font { size: 24 }
    effect: Reflection {
        topOffset:0
        fraction: 0.8
        topOpacity: 0.3
        bottomOpacity: 0.0
    }
}
```

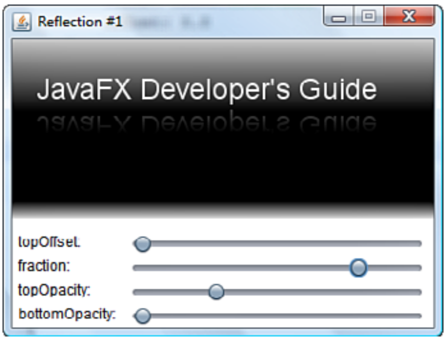

Figure 20-19 The **`Reflection`** effect

The `topOffset` variable lets you set the distance between the source object, here the text "JavaFX Developer's Guide" (and its reflection). Increasing this distance makes it seem that the source is further away from the reflecting surface. In Figure 20-19, the `topOffset` value is `0`, which places the reflection as close as possible to the original. In this case, the reflected text might seem to be farther away than it should be with this value—that is because of the descender on the letter *p*, which is the closest point of contact with the reflection.

The `fraction` variable determines how much of the source appears in the reflection. Typically, unless the reflecting surface is very shiny, you will not want the whole of the source object to be reflected. In Figure 20-19, the `fraction` variable has the value `0.8`, so about 80% of the source is reflected.

The `topOpacity` and `bottomOpacity` values give the opacity of the reflection at its top and bottom extents, respectively. In Figure 20-19, the `topOpacity` has been set to `0.3` and `bottomOpacity` to `0.0`.

SepiaTone

The `SepiaTone` effect is used to give images (or any group of nodes) an "Olde Worlde" look, as if they have been photographed by an old black-and-white camera, or washed out by the effects of exposure to sunlight over a long period. This effect provides only the two variables listed in Table 20-14.

Table 20-14 **Variables of the `SepiaTone` Class**

Variable	Type	Access	Default	Description
`input`	`Effect`	RW	`null`	The input to this effect
`level`	`Number`	RW	`1.0`	The level of this effect, from `0.0` to `1.0`

The `level` variable determines the extent to which the image is affected. The example code in the file `javafxeffects/SepiaTone1.fx` creates a `Scene` containing an image and a slider that lets you vary the value of the `level` variable and observe the result. The screenshot on the left of Figure 20-20 has `level` set to the value `0.4`, while the one on the right has `level` set to `1.0`.

PerspectiveTransform

The `PerspectiveTransform` is a useful effect that you can use to create the impression of a rotation in the direction of the z-axis—that is, into and out of the screen. It operates by deforming a node or group of nodes by moving its corners to specified locations and relocating the other pixels in such a way that straight lines drawn on the original nodes are mapped to straight lines in the result. Unlike the affine transforms that you saw in Chapter 17, "Coordinates, Transforms, and Layout," this effect does *not* guarantee that lines that are parallel in the source will be parallel in the result and, in fact, the perspective effect requires that some parallel lines be made nonparallel.

The variables that control the perspective effect are listed in Table 20-15.

Figure 20-20 The **`SepiaTone`** effect

Table 20-15 **Variables of the `PerspectiveTransform` Class**

Variable	Type	Access	Default	Description
`input`	`Effect`	RW	`null`	The input to this effect.
`llx`	`Number`	RW	`0`	The `x` coordinate of the location to which the lower-left corner of the input is moved
`lly`	`Number`	RW	`0`	The `y` coordinate of the location to which the lower-left corner of the input is moved
`ulx`	`Number`	RW	`0`	The `x` coordinate of the location to which the upper-left corner of the input is moved
`uly`	`Number`	RW	`0`	The `y` coordinate of the location to which the upper-left corner of the input is moved
`lrx`	`Number`	RW	`0`	The `x` coordinate of the location to which the lower-right corner of the input is moved
`lry`	`Number`	RW	`0`	The `y` coordinate of the location to which the lower-right corner of the input is moved
`urx`	`Number`	RW	`0`	The `x` coordinate of the location to which the upper-right corner of the input is moved
`ury`	`Number`	RW	`0`	The `y` coordinate of the location to which the upper-right corner of the input is moved

To see what these variables represent, refer to Figure 20-1. Here, imagine that the image is mounted vertically and can rotate about its vertical axis, as shown by the white dashed line. In the figure, the black outline represent the view of the image after it has been rotated a few degrees so that the right edge has moved closer to the viewer and the left edge farther away. This would cause the right edge to appear larger and the left edge correspondingly smaller, giving the impression of perspective.

You can use a `PerspectiveTransform` to create the rotated image from the original by moving the corners of the original to the new positions, as shown in Figure 20-21. The corner at the top left is the upper-left corner, and its position is given by the `ulx` and `uly` variables. The corner at the top right is the upper-right corner, and its position is specified by the `urx` and `ury` variables, and so on.

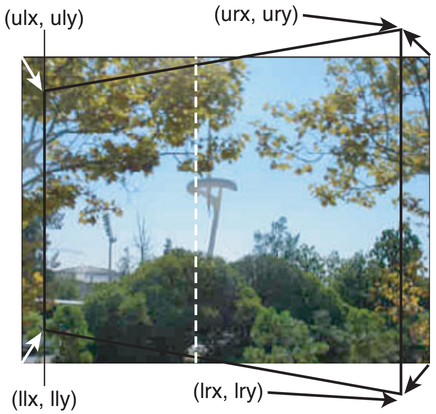

Figure 20-21 Illustrating the variables of the **`PerspectiveTransform`** class

It's easy to create a `PerspectiveTransform` that will rotate an image (or any other node or group) around a vertical axis that is a specified distance along its horizontal edge. It requires only a small amount of mathematics. We'll deal with the `x` and `y` coordinates separately, to make it easier to understand what is going on. The information needed to work out how to calculate the values of the `x` coordinates is shown in Figure 20-22.

Here, we are looking down at the image from above. The thick horizontal line, labeled APB, is the image before rotation, whereas the diagonal line, labeled A'PB', is the image after it has been rotated through an angle (shown here as `angle`) about a pivot point, marked P, that is l1 pixels from its left side and l2 pixels from its right side. In this case, the pivot point is almost equidistant from the sides of the image, but the same calculation works even if this is not the case. The x-axis is shown at the bottom of the figure.

The `x` coordinate of the left side of the image after rotation is given by the distance AC, while the `x` coordinate of the right side is given by A. The distance AC is the same as AP – CP. Because AP has the value `l1`, elementary trigonometry gives the following:

```
AC = AP - CP = l1 - l1 * cos(angle)
```

Similarly,

```
AD = BP + PD = l1 + l2 * cos(angle)
```

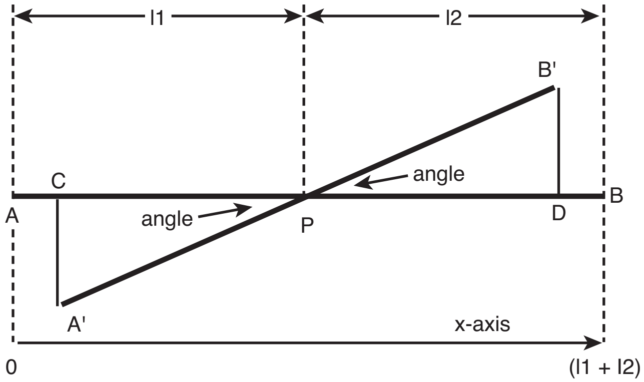

Figure 20-22 Rotating an image (top-down view)

AC is actually the value of both `ulx` and `llx`, while AD is the value that we need for `urx` and `ulx`. To make this simpler when using the `PerspectiveTransform`, we introduce two new parameters:

- `imgWidth`: The width of the image. This corresponds to the length AB and is equal to `l1 + l2`.
- `pivot`: The position of the pivot point along the line AB, as a ratio of `l1` to the total length AB. To place the pivot point in the center, set `pivot` to `0.5`.

Given these parameters, we so far have the following `PerspectiveTransform`:

```
PerspectiveTransform {
    ulx: l1 - l1 * Math.cos(angle)
    uly: ?? // Not yet determined
    llx: l1 - l1 * Math.cos(angle)
    lly: ?? // Not yet determined
    urx: l1 + l2 * Math.cos(angle)
    ury: ?? // Not yet determined
    lrx: l1 + l2 * Math.cos(angle)
    lry: imgHeight ?? // Not yet determined
}
```

Now let's move on to the y coordinates. This part is slightly easier. Essentially, what we need to do is make the length of the side of the image that moves toward us larger and that of the side that moves away from us smaller. We can choose by how much we want each side to grow or shrink—the closer we are to the image, the more each side would grow or shrink. We'll make this a parameter of the transform and say that we want each side to grow or shrink by `htFactor` of its actual value at each end. That means, for example, that if the image is 100 pixels tall and we choose `htFactor` to be `0.2`, the side of the image that is nearer to us after the image has rotated through 90 degree will be larger by 0.2 * 100 = 20 pixels at each end, or a total of 140 pixels tall. Similarly, the side that is farther away will shrink to 60 pixels in height.

Now refer to Figure 20-23. Here, we are looking at the image from the front again. The solid shape is the image after it has been rotated. The dashed vertical line is the axis of rotation, and the dashed extension that is outside the rectangle represents the maximum apparent height of the image when it has rotated through 90 degrees—that is, when it is edge-on to the viewer.

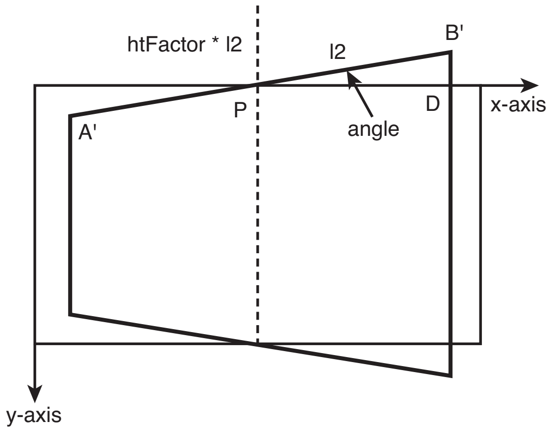

Figure 20-23 Rotating an image (front view)

In its current position, the y coordinate of the upper-right corner would be -B'D. This coordinate is negative because the y-axis runs along the top of the image, as shown. The length of B'D is `l2 * sin(angle)`, but because we are limiting the maximum vertical extension of each side by `htFactor`, we use the value `htFactor * l1 * sin(angle)` instead. Applying the same logic to each of the four corners gives us the following as the final transform, installed in an `ImageView` and with specific values assigned for the variables `pivot` and `htFactor`:

```
ImageView {
    translateX: bind (scene.width - imgWidth) / 2
    translateY: bind (scene.height - 30 - imgHeight) / 2
    image: image = Image { url: "{__DIR__}image1.jpg" }
    var angle = bind Math.toRadians(slider.value);
    var pivot = 0.5;
    var htFactor = 0.2;
    var l1 = bind pivot * imgWidth;
    var l2 = bind imgWidth - l1;
```

```
    effect: bind PerspectiveTransform {
        ulx: l1 - l1 * Math.cos(angle)
        uly: htFactor * l1 * Math.sin(angle)
        llx: l1 - l1 * Math.cos(angle)
        lly: imgHeight - l1 * htFactor * Math.sin(angle)
        urx: l1 + l2 * Math.cos(angle)
        ury: -l2 * htFactor * Math.sin(angle)
        lrx: l1 + l2 * Math.cos(angle)
        lry: imgHeight + l2 * htFactor * Math.sin(angle)
    }
}
```

The file `javafxeffects/PerspectiveTransform1.fx` contains an example that incorporates this transform and provides a slider that allows you to vary the value of the angle variable from –90 degrees to +90 degrees. Figure 20-24 shows a couple of screen-shots taken from this example with the image rotated by two different angular amounts. You can experiment with this example by changing the value of the `pivot` variable to get a rotation about a different point. Setting `pivot` to `0` causes a rotation around the left edge, while the value `1` gives rotation about the right edge.

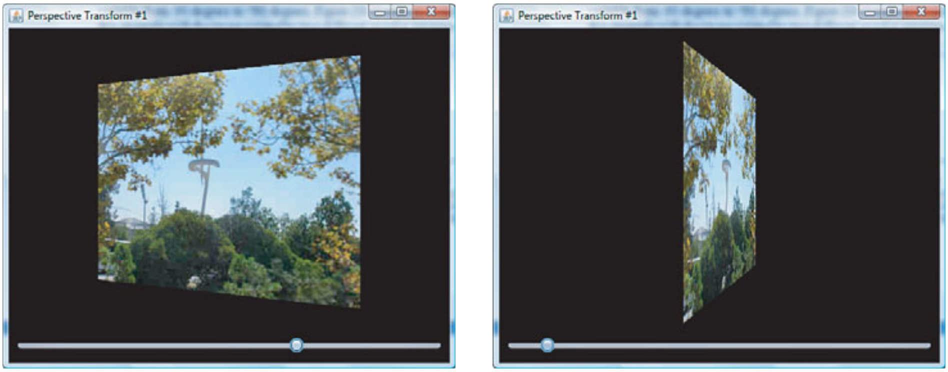

Figure 20-24 Examples of images rotated using a **`PerspectiveTransform`**

Note

You might be wondering why `PerspectiveTransform` is an effect and not one of the transforms discussed in Chapter 17. The reason is that it is not a true transform, in the sense that it does not affect the coordinate axes—it is just a visual effect. As a result of this, if you try to detect and act on mouse events from a node or group that has a `PerspectiveTransform` applied, you will not get reliable results because the coordinates in the event relate to the *untransformed* shape.

DisplacementMap

The `DisplacementMap` effect is, at first glance, the most complex of the effects that are provided by the JavaFX SDK, but it is also one of the most powerful. As its name suggests, this effect displaces pixels from their locations in the input image to different positions in the output image. Let's begin by listing the variables that you can use to parameterize the effect (see Table 20-16), and then we'll take a look at how they work.

Table 20-16 **Variables of the `DisplacementMap` Class**

Variable	Type	Access	Default	Description
`input`	`Effect`	RW	`null`	The input to this effect
`mapData`	`FloatMap`	RW	`Empty map`	The map that determines how input pixels are mapped to output pixels
`offsetX`	`Number`	RW	`0.0`	A fixed displacement along the x-axis applied to all pixel offsets
`offsetY`	`Number`	RW	`0.0`	A fixed displacement along the y-axis applied to all pixel offsets
`scaleX`	`Number`	RW	`1.0`	A scale factor applied to the map data along the x-axis
`scaleY`	`Number`	RW	`1.0`	A scale factor applied to the map data along the y-axis
`wrap`	`Boolean`	RW	`false`	Whether the displacement operation should wrap at the boundaries

How the `DisplacementMap` Effect Works

The reason for the apparent complexity of this effect is the equation that controls how the pixels are moved:

```
dst[x, y] = src[x + (offsetX + scaleX * map[x, y][0]) * srcWidth,
                y + (offsetY + scaleY * map[x, y][1]) * srcHeight]
```

At first sight, this probably looks quite daunting, but in fact it turns out to be quite simple. Basically, it says each pixel in the output (here represented by the symbol `dst`) derives from a single pixel in the input (represented by `src`). The pixel value at coordinates (`x`, `y`) in the output is obtained from a source pixel whose coordinates are displaced from those of the destination pixel by an amount that depends on a value obtained from a map, together with some scale factors and an offset. The values `srcWidth` and `srcHeight` are respectively the width and height of the input source.

Let's start by assuming that the offset values are both `0` and the scale values are both `1`. In this simple case, the equation shown above is reduced to this more digestible form:

```
dst[x, y] = src[x + map[x, y][0] * srcWidth,
                y + map[x, y][1] * srcHeight]
```

The map is a two-dimensional data structure that is indexed by the `x` and `y` coordinates of the destination point, relative to the top-left corner of the output image. Each element of this structure may contain a number of floats, which is why the class that holds these values is called a `FloatMap`. The `FloatMaps` that are used with a `DisplacementMap` must have two floats in each position,[5] the first of which is used to control the displacement along the x-axis and the second the displacement along the y-axis. Suppose, for the sake of argument, that we have a `FloatMap` in which every element has the values (–0.5, –0. 5). In this case, the equation above can be written as follows:

```
dst[x, y] = src[x - 0.5 * srcWidth, y - 0.5 * srcHeight]
```

Now, you should be able to see that the pixel at any given position in the output is obtained from the source pixel that is a half of the width or height of the source away from it. If we assume that the source is 100 pixels square, we can make our final simplification:

```
dst[x, y] = src[x - 50, y - 50]
```

This says that the output pixel at any point comes from the source pixel that is 50 pixels above it and to its left. The reason for using `srcWidth` and `srcHeight` as multipliers is that the values in the map can then be encoded as fractions of the width and height of the input respectively and therefore would normally be in the range -1 to +1. A map value of –1 or +1would move a point by the complete width or height of the input source.

A Simple Example

Let's look at how you would implement the example that you have just seen. You'll find the code in the file `javafxeffects/DisplacementMap1.fx`. Let's start by creating the map:

```
var image: Image = Image { url: "{__DIR__}image1.jpg" };
var imgWidth = image.width as Integer;
var imgHeight = image.height as Integer;
var map: FloatMap = FloatMap {
    width: imgWidth
    height: imgHeight
}

for (i in [0..<map.width]) {
    for (j in [0..<map.height]) {
        map.setSample(i, j, 0, -0.50);
        map.setSample(i, j, 1, -0.50);
```

[5] The `FloatMap` can have more than two floats in each position, but only the first two are used.

```
13            }
14        }
```

In this example, we are going to use an image as the input source, so we create a map that has the same dimensions as the image itself. The code on lines 4 to 6 declares the `FloatMap`, setting its dimensions from the width and height of the image. The nested loops on lines 9 to 14 initialize the `FloatMap`, assigning two samples for each element. Each sample has the value `-0.5`, which is the offset that we require. Note how these samples are installed:

```
map.setSample(i, j, 0, -0.50);   // The x offset
map.setSample(i, j, 1, -0.50);   // The y offset
```

`FloatMap` has several overloaded variants of the `setSample()` function that you can use. In the variant that we use here, the first two arguments are the `x` and `y` coordinates of the element, the third argument is the band number, and the fourth argument is the offset for that band. Band 0 is used for the x-offset and band 1 for the y offset.[6]

Now, here's the code that creates and uses the `DisplacementMap` effect:

```
var scene: Scene;
Stage {
    title: "DisplacementMap #1"
    scene: scene = Scene {
            width: 500
            height: 380
            fill: Color.BLACK
            content: [
                ImageView {
                    translateX: bind (scene.width - imgWidth) / 2
                    translateY: bind (scene.height - 30 - imgHeight) / 2
                    image: image
                    effect: DisplacementMap {
                        mapData: map
                    }
                }
            ]
        }
}
```

As you can see, the effect is applied simply by creating a `DisplacementMap` based on the map data and installing it in an `ImageView` that contains the source image. We don't need to set the scale or offset values because we are using the defaults in this case. You can see the result in Figure 20-25.

[6] The band numbers appear in the original equations. `map[x, y]`**[0]** indicates the value in band 0 at the element in position (x, y) in the map.

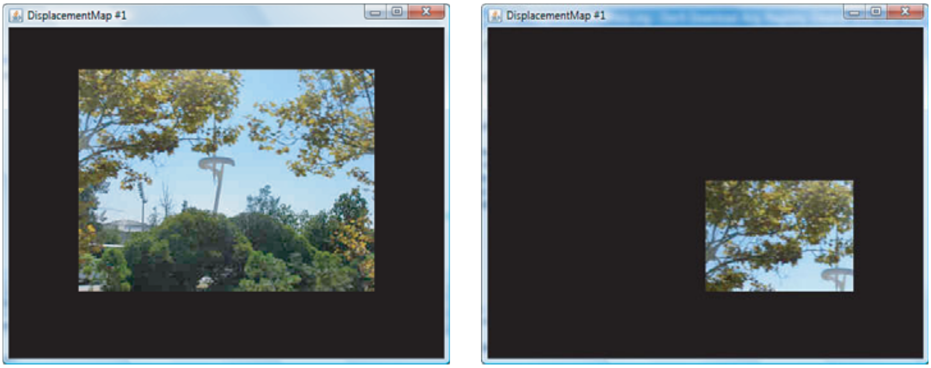

Figure 20-25 A simple **DisplacementMap** effect

The original image is shown on the left of the figure and the result of applying the `DisplacementMap` on the right. As you can see, the image has been moved halfway across and halfway down the area occupied by the source. It's easy to see why this has happened if you look back at the equation that describes this effect:

```
dst[x, y] = src[x - 0.5 * srcWidth,
                y - 0.5 * srcHeight]
```

This says that the pixel at (`x`, `y`) comes from the source pixel that is half the source width to its left and half the source height above it. In other words, the image is moved down and to the right. To make this more obvious still, let's add some concrete numbers. We'll start by with the pixel at (0, 0) in the destination image. According to the equation above, the color for this pixel comes from the pixel at (0 – 0.5 * 340, 0 – 0.5 * 255) = (–170, –127). Because there is no such point, this pixel is not set, so this part of the destination is transparent. In fact, every pixel for which either of the source coordinates is negative will be transparent. The first pixel in the destination image that will not be transparent is the one at (170, 127), which gets its color from the pixel at (0, 0) in the source. By following this reasoning for any given pixel in the destination image, it is easy to see why the result of this effect is to move the source down and to the right, as shown in Figure 20-25.

The `wrap` Variable

You can achieve a slightly different effect to that shown above by setting the `wrap` variable of the `DisplacementMap` object to `true`. When you do this, the parts of the destination that would have been transparent because they correspond to points in the source image that are outside of its bounds (for example, those with negative coordinates) are populated by wrapping the coordinates modulo the size of the source. This means, for example, that the pixel at (0,0), which should come from (–170, –127) in the source, will actually come from (–170 + 340, –127 + 128), or (170, 1). You can see the overall effect of this by running the code in the file `javafxeffects/DisplacementMap2.fx`, which gives the result shown in Figure 20-26.

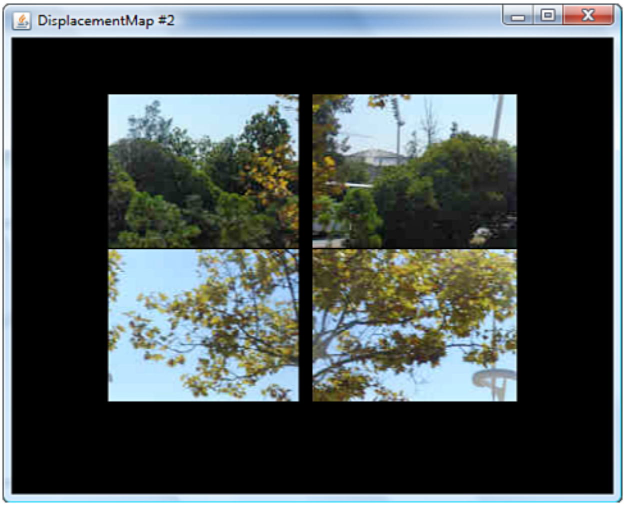

Figure 20-26 A **`DisplacementMap`** with wrap enabled

The `offset` Variables

Now that you have seen how the values in the map work in the simplest case, we'll make things a little more complex by adding back the `offsetX` and `offsetY` values. These values simply add a fixed offset to the distance between the destination pixel and the source pixel that supplies its color. Like the entries in the map, each offsets is scaled by the width or height of the source, as appropriate.

For example, let's suppose that we were to set the `offsetX` variable to `0.1` and leave `offsetY` as `0`. Then, using the same map as we did for the previous example, the equations that relate the source and destination pixel locations would now be as follows:

```
dst[x, y] = src[x + 0.1 * srcWidth- 0.5 * srcWidth,
                y - 0.5 * srcHeight]
```

If, as before, the source is 340 pixels wide, this change would produce an additional offset of 34 pixels between the source and destination pixels.

You can see how the `offsetX` value works by running the code in the file `javafxeffects/DisplacementMap3.fx`. This example uses the same `FloatMap` as the previous one, but adds a slider that allows you to vary the `offsetX` value from `0` up to `1.0`, with the initial value being `0.0`. Initially, the result looks the same as before, because the `offsetX` value is still `0`—compare the image on the left of Figure 20-27 with that on the right of Figure 20-25 to see that this is the case.

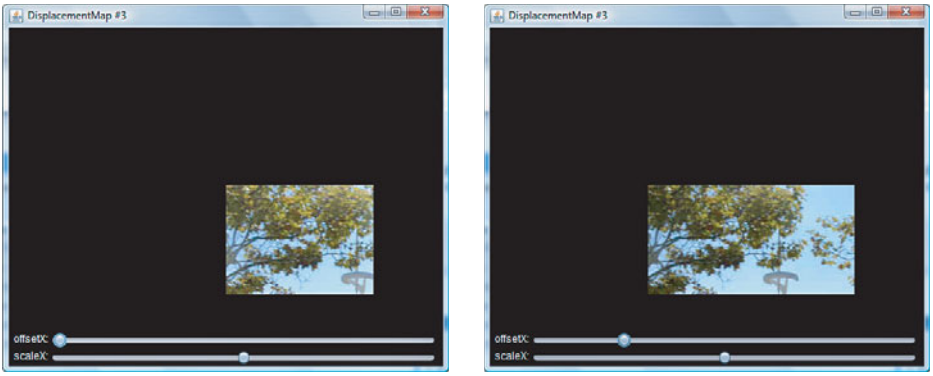

Figure 20-27 Varying the **`offsetX`** value of a **`DisplacementMap`** effect

Now if you move the offset slider to the right, you will see that the output image moves to the left. This is the offset at work. The farther you move the slider to the right, the more the result shifts to the left. The same effect would be seen along the y-axis if we had added a slider that allowed you to vary the `offsetY` value.

The `scale` Variables

The `scaleX` and `scaleY` variables are multipliers that are applied to the values from the `FloatMap`. If you use a `scaleX` value that is greater than 1, you make the offset between the source and destination pixels larger than that specified in the map. A `scaleX` value of `2` would double the offsets specified in the map. Similarly, if you use a value that is less than 1, the offset gets smaller. It is also possible to use a negative value, which would reverse the effect of the map.

The code in `javafxeffects/DisplacementMap3.fx` also includes a slider that lets you change the value of the `scaleX` variable over the range `0` to `2`, with `1` as its initial value. If you move the slider, you will find that the output image also moves to reflect the magnified or reduced offset values. In this case, because every entry in the map has the same value, the effect is very similar to that obtained by changing the `offsetX` value, but this is not always the case, as you'll see later in this section.

Using the `DisplacementMap` to Create a Warp

The example that we have been using has the same value in every element of the map. This is a rather unusual case and it doesn't produce a very interesting effect. In this section, we'll take a look at how to create a warp effect by populating the map with values that depend on their position in the map. The completed effect is shown in Figure 20-28.

As you can probably tell, the effect is produced by simulating the effect of a wave moving in the direction of the y-axis, which causes successive pixel rows to be displaced

Figure 20-28 Using **DisplacementMap** to create a warped effect

to the left or right of their initial positions. As there is no movement of any kind in the y direction, you can immediately conclude that all the y values (those in the second band) in the map are `0`. The wave effect is, in fact, a sine wave. Here's the code that populates the map[7]:

```
var image: Image = Image { url: "{__DIR__}image1.jpg" };
var imgWidth = image.width as Integer;
var imgHeight = image.height as Integer;
var map: FloatMap = FloatMap {
    width: imgWidth
    height: imgHeight
}

for (i in [0..<map.width]) {
    for (j in [0..<map.height]) {
        var value = (Math.sin(j/30.0 * Math.PI)/10;
        map.setSample(i, j, 0, value);
    }
}
```

[7] You'll find this code in the file `javafxeffects/DisplacementMap4.fx`.

The part that does all the interesting work is on line 11. It is obvious that this is creating a sine wave by supplying the horizontal displacement (in the first band of the map) for each row of the input source based on the value of the `Math.sin()` function. The value of this function varies from 0 at 0 radians to 1 at `PI/2` radians, back to zero at `PI` radians, to –1 at `3*PI/2` radians, and then back to zero at 2 `*PI` radians, and so on. In the inner loop, the value represents the pixel row. We divide it by 30 and multiply it by `PI` so that we get a complete wave over the space of 30 pixels. If you make this number larger, you will find that the wave spaces out more. This code would place values ranging from +1 to –1 in every element of the map. Remembering that these offsets are multiplied by the width of the source, this would mean that the image would be distorted by up to its full width. To reduce the distortion, we divide every value by 10, so we end up with values in the range –0.1 to +0.1. That's all we need to do to create a warp effect.

If you run the code in the file `javafxeffects/DisplacementMap4.fx`, you can use the offset and scale sliders to change the parameters of the `DisplacementMap`. Notice that changing the scale increases or decreases the amplitude of the sine wave, which results in more or less distortion.

Blending

Blending is the process of combining two pixels that would occupy the same space to produce a third value that is actually placed at that space. Blending can be used to determine what should be seen in a region where two nodes overlap or where two effects are applied to a node. You can use blending either as an effect or as a mode that controls the drawing of overlapping nodes in a group or container.

The `Blend` Effect

The `Blend` effect combines two inputs and produces a result that depends on the selected blend mode. The variables of the `Blend` class are listed in Table 20-17.

Here's an example that demonstrates how to construct a `Blend` effect. This code is extracted from the file `javafxeffects/BlendEffect1.fx`, which you can run to try out all the available blend modes:

```
var image1 = Image { url: "{__DIR__}image1.jpg" };
var image2 = Image { url: "{__DIR__}image2.jpg" };

ImageView {
    x: 30
    y: 30
    image: image1
    effect: Blend {
        mode: BlendMode.ADD
        topInput: Identity {
            x: 150
            y: 150
            source: image2
    }
}
```

Table 20-17 **Variables of the `Blend` Class**

Variable	Type	Access	Default	Description
`mode`	`BlendMode`	RW	`SRC_OVER`	The mode that determines how pixels from the two inputs are combined to produce the resulting pixel.
`topInput`	`Effect`	RW	`null`	The top input to this effect. If this is `null`, the node to which the effect is applied is used.
`bottomInput`	`Effect`	RW	`null`	The bottom input to this effect. If this is `null`, the node to which the effect is applied is used.
`opacity`	`Number`	RW	`1.0`	The opacity applied to the top input before blending.

Here, the bottom input is the `ImageView` itself (because the `bottomInput` variable is `null`, so the node itself becomes the input), while the top input is the output of an `Identify` effect applied to an `Image`. The second image is placed 150 pixels below and to the right of the `ImageView`, giving the result shown in Figure 20-29.

As you can see, the result of this effect is the union of the two images. There is a significant area of overlap between the two images, and in this region, their pixels are combined according to the unique set of rules that apply to the selected blend mode. There are 19 different modes, all defined as constants in the `BlendMode` class. You will find the details of each mode in the documentation for the `BlendMode` class. In Figure 20-29, `BlendMode.ADD` has been used. This adds all the color and alpha components from the two pixels to produce the result pixel. For example, if the RGBA values for two pixels were (0.6, 0.2, 0.3, 0.4) and (0.5, 0.6, 0.1, 0.5), the value of the resulting pixel would be (1.0, 0.8, 0.4, 0.9). Notice that the value of each channel is limited to `1.0`, which is why the result of combining the red channels in this example is `1.0` rather than `1.1`.

By selecting different values from the combo box at the top of the scene, you can see how each mode operates. Of particular interest are the `SRC_ATOP`, `SRC_IN`, `SRC_OUT` and `SRC_OVER` modes. In these modes, the "source" is the top input. You can see the results of applying `SRC_ATOP` mode on the left of Figure 20-30 and `SRC_IN` mode on the right.

The `SRC_ATOP` mode keeps all the bottom input plus that part of the top input that overlaps it. In the overlap area, only the top input is painted. By contrast, `SRC_IN` keeps only that part of the top input that overlaps with the bottom input, and everything else (including all the bottom input) is lost.

Figure 20-29 Using the **Blend** effect

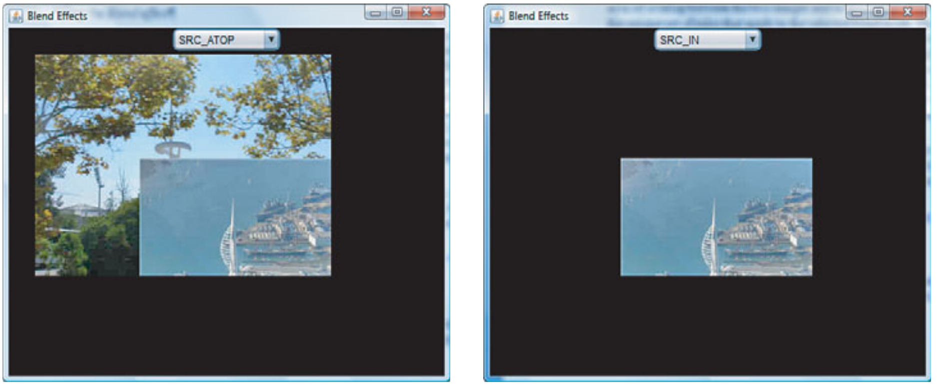

Figure 20-30 The **SRC_ATOP** and **SRC_IN** blend modes

The Group Blend Mode

A blend mode can be applied to a group (or a container, because `Container` is a subclass of `Group`) by setting its `blendMode` variable to one of the constants defined by the `BlendMode` class. The blend mode determines how the pixels in the areas in which there are overlapping nodes are constructed from those of the nodes themselves. By default, and in all the examples that you have seen so far, the `blendMode` variable has the value `BlendMode.SRC_OVER`, which causes the node in front to be drawn over those that are behind it.

The following code, from the file `javafxeffects/BlendGroup1.fx`, allows you to see how each of the possible blend modes operates when applied to the nodes of a group:

```
var image1 = Image { url: "{__DIR__}image1.jpg" };
var image2 = Image { url: "{__DIR__}image2.jpg" };
var scene: Scene;
Stage {
    title: "Blend Group"
    scene: scene = Scene {
            var modeCombo: SwingComboBox;
            var mode = bind modeCombo.selectedItem.value
                            as BlendMode;
            width: 500
            height: 400
            fill: Color.BLACK
            content: [
                modeCombo = SwingComboBox {
                    translateX: bind (scene.width
                       - modeCombo.layoutBounds.width) / 2
                    editable: false
                    items: [
                        SwingComboBoxItem {
                            value: BlendMode.ADD
                            text: "ADD"
                        }
                        // Further items not shown here
                    ]
                    selectedIndex: 0
                }
                Circle {
                    centerX: 100
                    centerY: 100
                    radius: 80
                    fill: Color.YELLOW
                }
                Group {
                    blendMode: bind if (mode != null) mode
                                    else BlendMode.ADD
                    content: [
                        ImageView {
                            x: 30
                            y: 30
                            image: image1
                        }
                        ImageView {
                            x: 150
```

```
                                    y: 150
                                    image: image2
                                }
                            ]
                        }
                    ]
                }
        }
```

This code creates a scene containing a combo box that allows a `BlendMode` to be selected, a circle, and a group containing two overlapping two `ImageViews`. On the left of Figure 20-31, you can see the result of applying the default mode, which is `SRC_OVER`. On the right of the figure, the `MULTIPLY` mode is used.

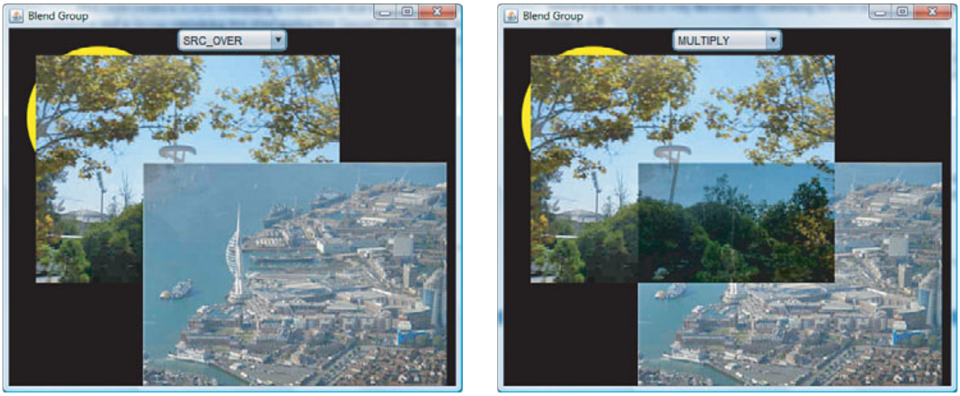

Figure 20-31 Using blend modes in a group

As you can see, these two modes have different effects on the pixels in the area of overlap between the two images. Note, however, that the blend mode has no effect on the region in which the upper-left `ImageView` overlaps the circle, because the circle is not in the group and therefore is not subject to the blend mode.

Lighting

The final effect that we are going to look at is called `Lighting`. As its name suggests, it allows you to specify how a node or a group should be lit. Three different types of lighting can be used, all of which are represented by classes in the `javafx.scene.effect.light` package. The `Lighting` class itself, like all the other effects, is in the `javafx.scene.effect` package.

To define the lighting for a node or group, you create an instance of the `Lighting` class and install it in the `effect` variable of that node or group. Using lighting will make your scenes look more three-dimensional than they otherwise would, as you'll see in the examples in this section. The variables of the `Lighting` class are listed in Table 20-18.

Table 20-18 **Variables of the `Lighting` Class**

Variable	Type	Access	Default	Description
`light`	`Light`	RW	`DistantLight`	The type of light to be used.
`bumpInput`	`Effect`	RW	`null`	The bump map to be applied.
`contentInput`	`Effect`	RW	`null`	The input to this effect, which is the target node itself if this value is `null`.
`diffuseConstant`	`Number`	RW	`1.0`	Determines how much diffuse light is reflected from the surface, in the range `0` to `2`, inclusive.
`specularConstant`	`Number`	RW	`0.3`	Determines how much specular light is reflected from the surface, in the range `0` to `2`, inclusive.
`specularExponent`	`Number`	RW	`20`	Determines how shiny the surface appears to be in the range `0` to `40`, inclusive.
`surfaceScale`	`Number`	RW	`1.5`	Determines the height assigned to pixels in the source, based on their opacity. Valid range is `0` to `10`, inclusive.

The `light` variable determines the type of lighting required, which must be one of `DistantLight` (which is the default), `PointLight`, and `SpotLight`. The other variables set the characteristics of the surface that will be lit and will be explained in the rest of this section by reference to examples.

The `surfaceScale` Variable

To create a 3D effect when lighting a 2D surface, the opacity value of each pixel is used as a guide to how "high" that pixel should appear to be when lit. Transparent pixels appear to be the lowest, while fully opaque pixels appear to be raised up from the surface. This effect can be increased or decreased by using the `surfaceScale` variable. Values that are greater than `1` cause the effect to be increased, while values between `0` and `1` cause it to be decreased. The same effect can also be seen at the edges of shapes.

The following code, which comes from the file `javafxeffects/SurfaceScale.fx`, shines a distant light on a rectangle. The details of the lighting are not important right now, but you'll see that the `surfaceScale` variable is bound to a slider. If you run this example, you can see the effect of varying the `surfaceScale` variable over its full range:

```
Rectangle {
    x: 20
    y: 20
    width: 100
    height: 100
    fill: Color.YELLOW
    effect: Lighting {
        light: DistantLight {
            azimuth: 0
            elevation: 30
        }
        surfaceScale: bind (scaleSlider.value as Number) / 10
    }
}
```

You can see how the `surfaceScale` value is used by comparing the two screenshots shown in Figure 20-32.

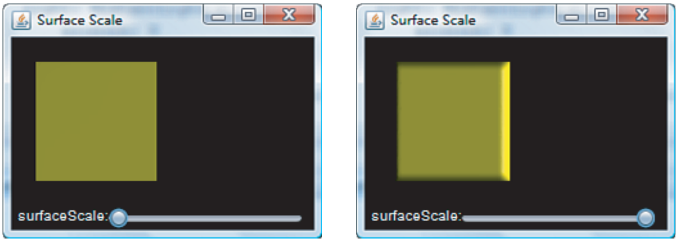

Figure 20-32 The **`surfaceScale`** variable of the **`Lighting`** effect

On the left of the figure, the `surfaceScale` value is `0`, so there is no 3D effect at all. On the right, `surfaceScale` has its maximum possible value, and now you can see that the center of the rectangle appears to be raised up above its edges. The higher the `surfaceScale` value, the higher the center will appear to be.

The Bump Map

You can add additional surface relief by creating a *bump map* and installing it in the `bumpInput` variable of the `Lighting` effect. If this variable is `null`, the node on which the effect is applied is itself used to generate a bump map, which is what causes the 3D effect at the edges that you saw in Figure 20-32.

The bump map is just an `Effect`, which supplies pixels from which the relief of the lit surface is calculated. As before, the apparent height of a pixel on the lit shape depends on the opacity of the corresponding pixel of the bump map. This effect is affected by the value of the `surfaceScale` variable, as described in the preceding section.

A common way to specify a bump map is to create an image, set the opacity to reflect the contours that you want to appear in the finished result, and then turn it into an effect by using the `Identity` class that we discussed earlier in this chapter. The code in the file `javafxeffects/BumpMap.fx` shows how to apply a bump map in the form of an image file:

```
var logo = Image { url: "{__DIR__}javafxlogo.gif" };
Stage {
    title: "Bump Map"
    scene: Scene {
        width: 240
        height: 140
        fill: Color.BLACK
        content: [
            Rectangle {
                x: 20
                y: 20
                width: 200
                height: 100
                fill: Color.YELLOW
                effect: Lighting {
                    light: DistantLight {
                        azimuth: 90
                        elevation: 25
                    }
                    bumpInput: Identity {
                        source: logo
                        x: 30
                        y: 40
                    }
                }
            }
        ]
    }
}
```

The code on line 1 loads the bump image from a file called `logo.gif` that is in the same directory as the script file. This image is then converted to an `Effect` by the code on lines 20 to 24, and placed appropriately relative to the rectangle node to which the lighting effect is applied by using the `x` and `y` variables of the `Identity` class. The lighting itself is a `DistantLight`, to which the image is supplied as the bump map on line 20.

The image that is used as the bump map is shown on the left of Figure 20-33. The image consists of the word JavaFX in black text on a white background. The white background is actually completely transparent, while the black text is completely opaque. You can see the effect of the bump map on the right of Figure 20-33, where the lighting effect causes the word JavaFX to appear to be raised above the surface of the `Rectangle`.

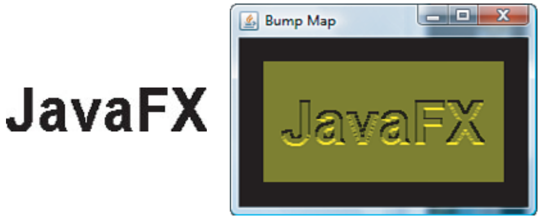

Figure 20-33 Using a bump map with a **`Lighting`** effect

You can apply the same effect to any node, including an `ImageView`, where you can use it to create the appearance of a watermark within the image.

DistantLight

The `DistantLight` class is used when you want to apply a more-or-less uniform light to a node or group. Depending on where the light source is, you may see some shadows, but you will not see reflections of the type that are a characteristic of `PointLight` and `Spotlight`, which are discussed in the sections that follow. `DistantLight`, `PointLight`, and `Spotlight` are all derived from the base class `javafx.scene.effect.light.Light`, which has a single variable called `color` (of type `Color`) that specifies the color of the light to be used, which is white by default. The other variables of the `DistantLight` class specify the position of the light source and are listed in Table 20-19.

The `azimuth` is the position of the light source on the plane of the scene. An `azimuth` angle of 0 degrees places the light source at the 3 o'clock position, one of 90 degrees moves it to 6 o'clock, and so on. Negative angles can also be used and are measured counterclockwise. For example, setting the `azimuth` variable to either `-90` or `270` places the light source at 12 o'clock.

Table 20-19 **Variables of the `DistantLight` Class**

Variable	Type	Access	Default	Description
`azimuth`	`Number`	RW	`45`	The azimuth of the light source, in degrees
`elevation`	`Number`	RW	`45`	The elevation of the light source, in degrees

The `elevation` gives the angle of the light source above or below the plane of the scene. When the `elevation` is `0` or `180`, the light source is on the plane of the scene, when it is `90`, it is overhead the scene and shining directly down on it, and when it is `270` (or `-90`), it is directly below the scene.

The following code, which you will find in the file `javafxeffects/DistantLight1.fx`, allows you to move a `DistantLight` source around a large yellow circle to see the effect that is created:

```
Circle {
    centerX: 200
    centerY: 180
    radius: 150
    fill: Color.YELLOW
    effect: Lighting {
        light: DistantLight {
            azimuth: bind azimuthSlider.value
            elevation: bind elevationSlider.value
        }
        surfaceScale: 5
    }
}
```

Figure 20-34 shows two different configurations of the `DistantLight` source.

On the left of the figure, the `azimuth` and `elevation` variables both have the value `45`, which places the light source at approximately the 4.30 position and elevated 45 degrees above its surface. You can see that this is the case because the lower-right edge of the circle is much brighter than the rest of it. On the right, the light source has been moved to the 9 o'clock position by setting the azimuth variable to `180` and moved very close to the plane of the scene as a result of the elevation, which is very nearly 0 degrees. Because of the low elevation, most of the circle is quite dark, with the exception of the edge at around the 9 o'clock position, which is closest to the light source.

It is worth examining here the effect of the `diffuseConstant` of the `Lighting` class. This constant acts as a multiplier to the RGB values of all the pixels on the lit surface. Therefore, you can use this variable to make the surface lighter or darker. The example in

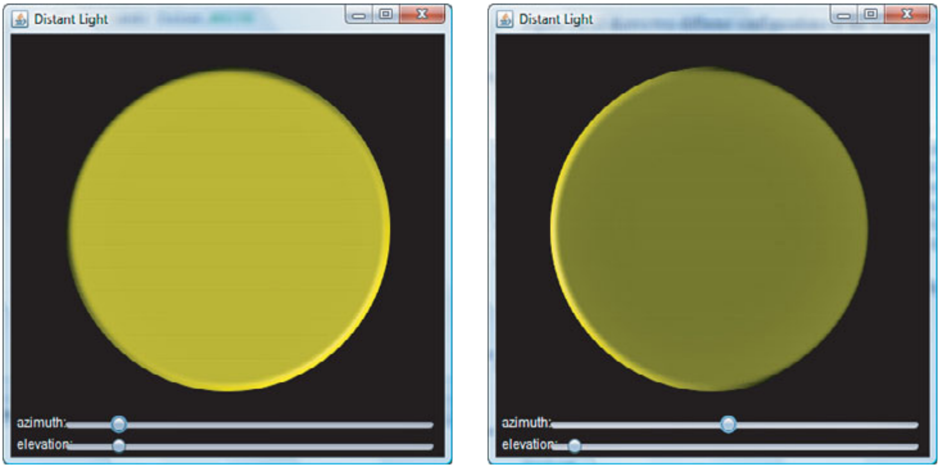

Figure 20-34 Using a **`DistantLight`** source

the file `javafxeffects/DistantLight2.fx` illustrates this by setting the `diffuseConstant` value of the `Lighting` effect to `1.5`, which has the result of making the circle brighter, as you can see by comparing the result shown in Figure 20-35 with Figure 20-34, where this variable had the value `1`.

Figure 20-35 The effect of the **`diffuseConstant`** on a lit surface

PointLight

`PointLight` represents a single point of light that is positioned somewhere relative to the surface to be lit. The variables of the `PointLight` class, as shown in Table 20-20, allow you to specify exactly where the light source should be placed.

Table 20-20 **Variables of the `PointLight` Class**

Variable	Type	Access	Default	Description
`x`	`Number`	RW	`0`	The `x` coordinate of the light source
`y`	`Number`	RW	`0`	The `y` coordinate of the light source
`z`	`Number`	RW	`0`	The `z` coordinate of the light source

In the following code, a `PointLight` source whose position is bound to the values of three sliders is created and applied to a large yellow circle. If you run this example, which can be found in the file `javafxeffects/PointLight1.fx`, you can experiment with the effect of changing the location of the light source:

```
Circle {
    centerX: 200
    centerY: 180
    radius: 150
    fill: Color.YELLOW
    effect: Lighting {
        light: PointLight {
            x: bind xSlider.value
            y: bind ySlider.value
            z: bind zSlider.value
        }
        surfaceScale: 5
        specularConstant: bind (specCSlider.value as Number) / 10
        specularExponent: bind specESlider.value
    }
}
```

You can see two different `PointLight` configurations in Figure 20-36. On the left, the light is at (`x` = `45`, `y` = `45`, `z` = `45`), which is to the top left of the circle itself. You can see that a `PointLight` source results in a more concentrated area of illumination than a `DistantLight`. On the right of the figure, the light source has been moved so that its reflection has moved more toward the center of the circle.

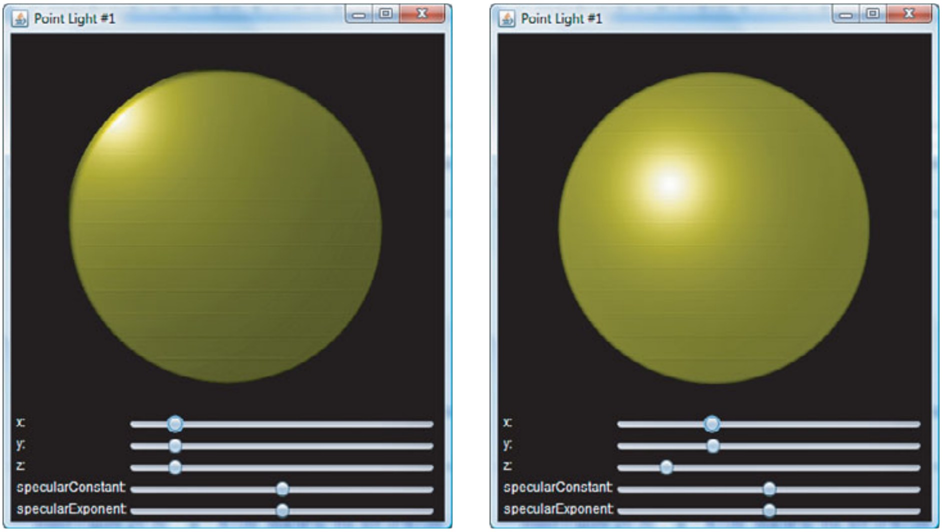

Figure 20-36 Using a **PointLight** source

The size and intensity of the reflection depends on values of the `specularConstant` and `specularExponent` variables of the `PointLight` class. Like `diffuseConstant`, `specularConstant` is a multiplier that is applied to the RGB values of the lit source, so values greater than `1` make the reflection bright, while values less than `1` make it dimmer. The `specularExponent` controls the spread of the light and therefore the radius of the reflected area. Increasing values of `specularExponent` reduce this radius and therefore make the reflection brighter. You can see examples that use different settings for these variables in Figure 20-37.

SpotLight

`PointLight` represents a single-point source of light that shines uniformly in every direction, much like the sun. `SpotLight` is a subclass of `PointLight` that acts like a point light source that radiates light over a more confined area. The rays of light are confined to the inside of a cone with its tip at the source. The axis of the cone points to a specified location on the surface of the object being lit. The combination of the position of the light source, the point at which it is aimed, and the width of the cone at the point at which the light reaches the lit object determines the lighting effect that you see. You can specify these values using the variables listed in Table 20-21.

The following code, which is from the file `javafxeffects/SpotLight1.fx`, applies a `SpotLight` effect to the same circle that we illuminated with a `PointLight` source in the previous example. The light source is placed 60 pixels above the center of the circle, while

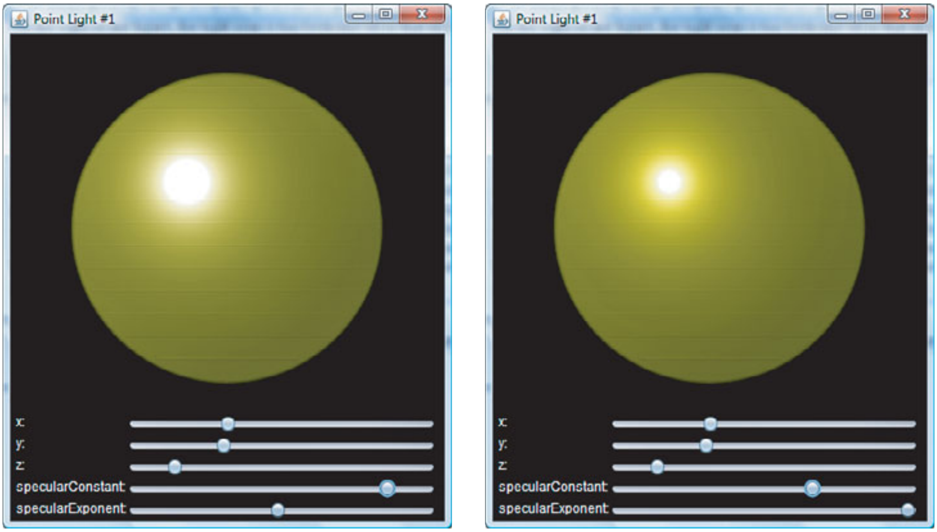

Figure 20-37 The effects of the **`specularConstant`** and **`specularExponent`** variables

Table 20-21 **The Variables of the `SpotLight` Class**

Variable	Type	Access	Default	Description
`pointsAtX`	`Number`	RW	`0.0`	The `x` coordinate of the point at which the light is aimed
`pointsAtY`	`Number`	RW	`0.0`	The `y` coordinate of the point at which the light is aimed
`pointsAtZ`	`Number`	RW	`0.0`	The `z` coordinate of the point at which the light is aimed
`specular Exponent`	`Number`	RW	`1.0`	Controls the width of the light cone, in the range `0` to `4`, inclusive

the point at which it is aimed can be controlled by three sliders. The effect produced by the `SpotLight` with these initial variable settings is shown on the left of Figure 20-38.

```
Circle {
    centerX: 200
    centerY: 180
    radius: 150
    fill: Color.YELLOW
    effect: Lighting {
        light: SpotLight {
            pointsAtX: bind xSlider.value
            pointsAtY: bind ySlider.value
            pointsAtZ: bind zSlider.value
```

```
            x: 200
            y: 180
            z: 60
            specularExponent: bind (specESlider.value as Number)/10
        }
        surfaceScale: 5
    }
}
```

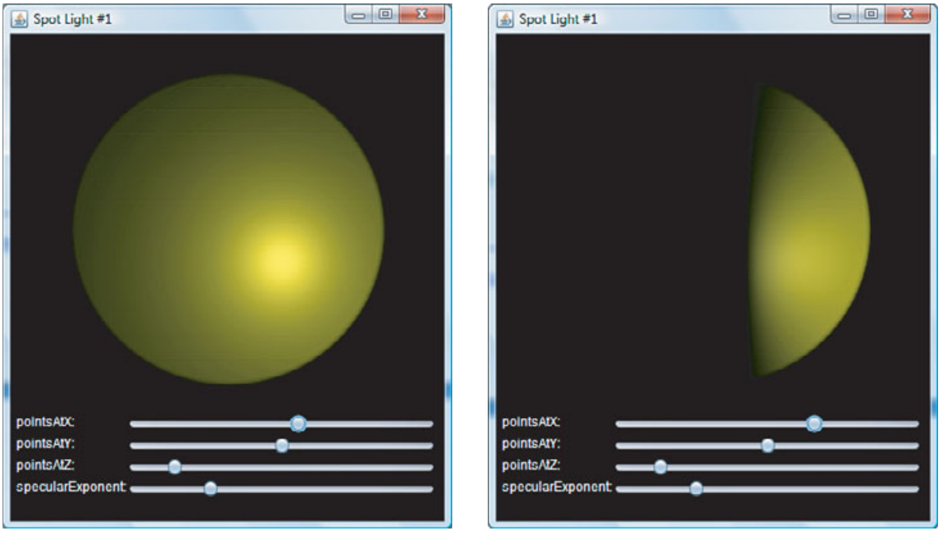

Figure 20-38 The **`SpotLight`** effect

Moving the aiming point changes the resultant lighting effect. When the light source is quite close to the target object, as it is in this case, even a small change in the aiming point can make a noticeable difference to the result. On the right of Figure 20-38, the aiming point has been moved only a small amount to the right and, as you can see, almost half of the circle is now in darkness.

The bottom slider in Figure 20-38 allows you to see the effect of changing the `specularExponent` value,[8] which is initially set to its default value of `1`. Increasing this value makes the light cone narrower that produces a more focused beam and therefore a smaller and brighter effect on the target, as you can see in Figure 20-39, where this variable has its maximum value of `4.0`. With this setting, almost all the light is confined to a small area around the aiming point.

---

[8] Do not confuse this variable with the `specularExponent` variable of the `Lighting` class.

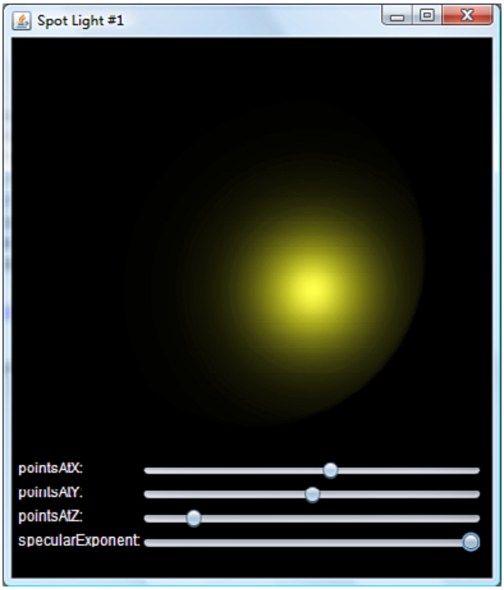

Figure 20-39 The effect of the **`specularExponent`** variable of the **`SpotLight`** class

Index

Symbols and Numerics

A

C

D

F

G

J

K

M

N

O

P

Q

R

S

U

V

X

Y

Z