5 Behind the scene graph

This chapter covers

- Defining a scene graph
- Animating stuff on screen, with ease
- Transforming graphics and playing with color
- Responding to mouse events

In chapter 4 we looked at building a rather traditional UI with Swing. Although Swing is an important Java toolkit for interface development, it isn't central to JavaFX's way of working with graphics. JavaFX comes at graphics programming from a very different angle, with a focus more on free-form animation, movement, and effects, contrasting to Swing's rather rigid widget controls. In this chapter we'll be taking our first look at how JFX does things, constructing a solid foundation onto which we can build in future chapters with evermore sophisticated and elaborate graphical effects.

The project we'll be working on is more fun than practical. The idea is to create something visually interesting with comparatively few lines of source code—certainly far fewer than we'd expect if we were forced to build the same application using a language like Java or C++. One of the driving factors behind JFX is to allow rapid prototyping and construction of computer visuals and effects, and it's

this speed and ease of development I hope to demonstrate as we progress through the chapter.

We'll be exploring what's known as the *scene graph*, the very heart of JavaFX's graphics functionality. We touched on the scene graph briefly in the last chapter; now it's time to get better acquainted with it. The scene graph is a remarkably different beast than the Java2D library typically used to write Java graphics, but it's important to remember that one is not a replacement for the other. Both JavaFX's scene graph and Java2D provide different means of getting pixels on the screen, and each has its strengths and weaknesses. For slick, colorful visuals the scene graph model has many advantages over the Java2D model—we'll be seeing exactly why that is in the next section.

5.1 What is a scene graph?

There are two ways of looking at graphics: a blunt, low-level "throw the pixels on the screen" approach and a higher-level abstraction that sees the display as constructed from recognizable primitives, like lines, rectangles, and bitmap images.

The first is what's called an *immediate mode* approach, and the second a *retained mode* approach. In immediate mode each element on the display is instructed to draw itself into a designated part of the display immediately—no record is kept of what is being drawn (other than the destination bitmap itself, of course). By comparison, in retained mode a tree structure is created detailing the type of graphic elements resident on the display—the upkeep of this (rendering it to screen) is no longer the responsibility of each element.

Figure 5.1 is a representation of how the two systems work.

We can characterize the immediate mode approach like so: "When it's time to redraw the screen I'll point you toward your part of it, and you take charge of drawing what's necessary." Meanwhile the retained mode approach could be characterized as follows: "Tell me what you look like and how you fit in with the other elements on the display, and I'll make sure you're always drawn properly, at the correct position, and updated when necessary."

This offloading of responsibility allows any code using the retained mode model to concentrate on other things, like animating and otherwise manipulating its elements, safe in the knowledge that all changes will be correctly reflected on screen.

So, what is a scene graph? It is, quite simply, the structure of display elements to be maintained onscreen in a retained mode system.

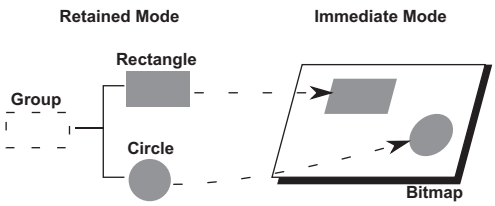

Figure 5.1 A symbolic representation of retained mode and immediate mode. The former sees the world as a hierarchy of graphical elements, the latter as just pixels.

5.1.1 Nodes: the building blocks of the scene graph

The elements of the scene graph are known as *nodes*. Some nodes describe drawing primitives, such as a rectangle, a circle, a bitmap image, or a video clip. Other nodes act as grouping devices; like directories in a filesystem, they enable other nodes to be collected together and a treelike structure to be created. This treelike structure is important for deciding how the nodes appear when they overlap, specifically which nodes appear in front of other nodes and how they are related to one another when manipulated. We can best demonstrate this functionality using figure 5.2.

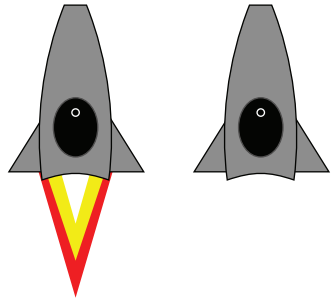

Figure 5.2 Elements in a scene graph can be manipulated without concern for how the actual pixels will be repainted. For example, hiding elements will trigger an automatic update onscreen.

A rocket ship might be constructed from several shapes: a distorted rectangle for its body, two triangular fins, and a black, circular cockpit window. It may also have a little rocket jet pointing out of its tail, likewise constructed from shapes. Each shape would be one primitive on the scene graph, one node in a tree-like structure of elements that can be rendered to screen. When nodes are manipulated, such as toggling the rocket jet to simulate a flickering flame, the display is automatically updated.

5.1.2 Groups: graph manipulation made easy

Once shapes have been added to a scene graph, we can manipulate them using such transformations as a rotation. But the last thing we want is for the constituent parts to stay in the same location when rotated. The effect might be a tad unsettling if the fins on our rocket ship appeared to fly off on an adventure all their own (figure 5.3). We want the whole ship to rotate consistently, as one, around a single universal origin.

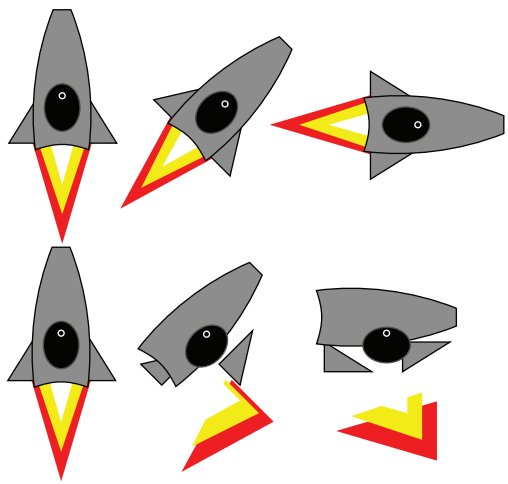

Figure 5.3 Grouping nodes in a scene graph allows them to be manipulated as one. The upper rocket has been rotated as a group; the lower rocket has been rotated as separate constituent nodes.

Groups are the answer! They allow us to combine several scene graph elements so they can be manipulated as one. Groups can be used within groups to form a hierarchy; the rocket's body and flame could be grouped separately within a main group, allowing the latter to be toggled on or off by flipping its visibility, as shown in figure 5.2.

This introductory text has only brushed the surface of the power scene graphs offer us. The retained mode approach allows sophisticated scaling, rotation, opacity (transparency), filter, and other video effects to be applied to entire swathes of objects all at once, without having to worry about the mechanics of rendering the changes to screen.

So that's all there really is to the scene graph. Hopefully your interest has been piqued by the prospect of all this pixel-pushing goodness; all we need now is a suitable project to have some fun with.

5.2 Getting animated: LightShow, version 1

The eighties were a time of massive change in the computer industry. As the decade began, exciting new machines, such as Space Invaders and Pac-Man, were already draining loose change from the pockets of unsuspecting teenage boys, and before long video games entered the home thanks to early consoles and microcomputers. An explosion in new types of software occurred, some serious, others just bizarre.

Pioneered by the legendary llama-obsessed games programmer Jeff Minter, Psychedelia (later, Colourspace and Trip-a-Tron) provided strange real-time explosions of color on computer monitors, ideal for accompanying music. The concept would later find its way into software like Winamp and Windows Media Player, under the banner of *visualizations*.

In this chapter we're going to develop our own, very simple *light synthesizer*. It won't respond to sound, as a real light synth should, but we'll have a lot of fun throwing patterns onscreen and getting them to animate—a colorful introduction (in every sense) to the mysterious world of JavaFX's scene graph.

At the end of the project you should have a loose framework into which you can plug your own scene graph experiments. So let's plunge in at the deep end by seeing how to plug nodes together.

5.2.1 Raindrop animations

The JavaFX scene graph API is split into many packages, specializing in various aspects of video graphics and effects. You'll be glad to know we'll be looking at only a handful of them in this chapter. At its heart, the scene graph centers on a single element known as a *node*. There are numerous nodes provided in the standard API; some draw shapes, some act as groups, while others are concerned with layout. All the nodes are linked, at the top level, into a *stage*, which provides a bridge to the outside world, be that a desktop window or a browser applet.

For our light synthesizer we're going to start by creating a raindrop effect, like tiny droplets of water falling onto the still surface of a pond. For those wondering (or perhaps *pondering*) how this might look, the effect is caught in action in figure 5.4.

Figure 5.4 Raindrops are constructed from several ripples. Each ripple expands outward, fading as it goes.

Before we begin, it's essential to pin down exactly how a raindrop works from a computer graphics point of view:

- Each raindrop is constructed from multiple ripple circles.
- Each ripple circle animates, starting at zero width and growing to a given radius, over a set duration. As each ripple grows, it also fades.
- Ripples are staggered to begin their individual animation at regular beats throughout the lifetime of the overall raindrop animation.

Keen-eyed readers will have spotted two different types of timing going on here: at the outermost level we have the raindrop activating ripples at regular beats, and at the lowest level we have the smooth animation of an individual ripple running its course, expanding and fading. These are two very different types of animation, one digital in nature (jumping between states, with no midway transitions) and the other analog in nature (a smooth transition between states), combining to form the overall raindrop effect.

5.2.2 The RainDrop class: creating graphics from geometric shapes

Now that you know what we're trying to achieve, let's look at a piece of code that defines the scene graph. See listing 5.1.

Listing 5.1 RainDrop.fx

```
package jfxia.chapter5;

import javafx.animation.Interpolator;
import javafx.animation.KeyFrame;
import javafx.animation.Timeline;
import javafx.lang.Duration;
import javafx.scene.Group;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;

package class RainDrop extends Group {
    public-init var radius:Number = 150.0;
    public-init var numRipples:Integer = 3;
    public-init var rippleGap:Duration = 250ms;
    package var color:Color = Color.LIGHTBLUE;

    var ripples:Ripple[];
    var masterTimeline:Timeline;

    init {
        ripples = for(i in [0..<numRipples]) Ripple {
            stroke: bind color;
            animRadius: radius;
        };
        content = ripples;
        masterTimeline = Timeline {
            keyFrames:
                for(i in [0..<numRipples]) KeyFrame {
                    time: i*rippleGap;
                    action: function() {
```

Subclasses Group

External interface variables

Multiple Ripple instances

Timeline to activate ripples

```
                    ripples[i].rippleTimeline
                        .playFromStart();
                }
            };
        };
    }

    package function start(x:Integer,y:Integer) : Void {
        this.layoutX = x;
        this.layoutY = y;
        masterTimeline.playFromStart();
    }
}

class Ripple extends Circle {
    var animRadius:Number;
    override var fill = null;

    def rippleTimeline = Timeline {
        keyFrames: [
            at (0ms) {
                radius => 0;
                opacity => 1.0;
                strokeWidth => 10.0;
                visible => true;
            } ,
            at (1.5s) {
                radius => animRadius
                    tween Interpolator.EASEOUT;
                opacity => 0.0
                    tween Interpolator.EASEOUT;
                strokeWidth => 5.0
                    tween Interpolator.LINEAR;
                visible => false;
            }
        ]
    };
}
```

Timeline to activate ripples

Starts animating ripples

Subclasses Circle

Start animation state

Finish animation state

Listing 5.1 creates two classes, `Raindrop` and `Ripple`. Together they form our desired raindrop effect onscreen, with multiple circles fanning out from a central point, fading as they go. The code will not run on its own—we need another bootstrap class, which we'll look at in a moment. For now let's consider how the raindrop effect works and how the example code implements it.

The second class, `Ripple`, implements a single animating ripple, which is why it subclasses the `javafx.scene.shape.Circle` class. Each circle is a node in the scene graph, a geometric shape that can be rendered onscreen. A raindrop with just one ripple would look rather lame. That's why the first class, `RainDrop`, is a container for several `Ripple` objects, subclassing `javafx.scene.Group`, which is the standard JavaFX scene graph group node.

The `Group` class works like the `Flow` class we encountered last chapter, except it does not impose any layout on its children. The `content` attribute is a sequence of

Node objects, which it will draw, from first to last, such that earlier nodes are drawn below later ones.

Child nodes are positioned within their parent Group using the layoutX and layoutY variables inherited from Node, which is the aptly named parent class of all scene graph node objects. Circle objects use their center as a coordinate origin, while other shapes (like Rectangle) might use their top-left corner. Coordinates are local to their parent, as figure 5.5 explains. The actual *onscreen* coordinates of a given node are the sum of its own layout translation plus all layout translations of its parent groups, both direct and indirect.

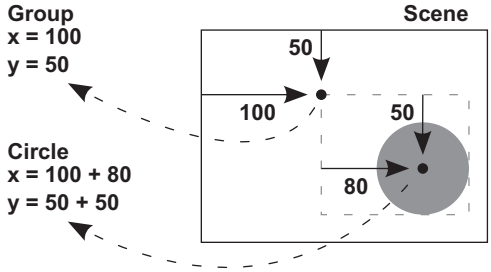

Figure 5.5 Groups provide a local coordinate space for their children. The Group is laid out to (100,50) and the Circle (positioned around its center) to (80,50), giving an absolute position of (180,100).

Enough of groups—what about our code? We'll study the animation inside Ripple shortly, but first we need to understand the container class, RainDrop, where the raindrop's external interface lies.

First we define public-init variables, allowing other classes to manipulate our raindrop declaratively. The radius is the width each ripple will grow to, while numRipples defines the number of ripples in the overall raindrop animation, and rippleGap is the timing between each ripple being instigated. Finally color is, unsurprisingly, the color of the ripple circles. Later in the project we're going to manipulate the raindrop hue, so we've made color externally writable.

The private variable ripples holds our Ripple objects. You can see it being set up in the init block and then plugged into the scene graph via content in (parent class) Group.

Another private variable being set up in init is masterTimeline, which fires off each individual ripple circle animation at regular beats, controlled by rippleGap. The remainder of the class is a function that activates this animation. The function moves RainDrop to a given point, around which the ripples will be drawn, and kicks off the animation.

Now all we need to know is how the animation works.

5.2.3 Timelines and animation (Timeline, KeyFrame)

Animation in JavaFX is achieved through *timelines*, as represented by the appropriately named Timeline class. A timeline is a duration into which points of change can be defined. In JavaFX those points are known as *key frames* (note the KeyFrame class reference in listing 5.1), and they can take a couple of different forms.

The first form uses a function type to assign a piece of code to run at a given point on the timeline, while the second changes the state of one or more variables across the duration between key frames, as represented in figure 5.6 (think back to the end of section 5.2.1 when we discussed digital- and analog-style animations).

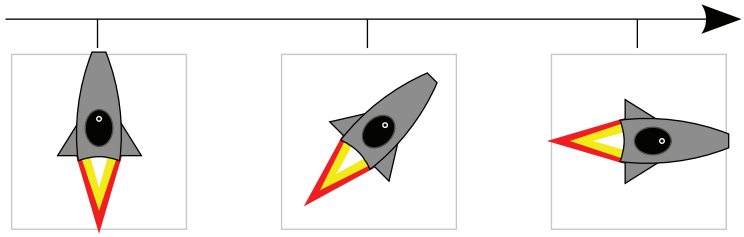

Figure 5.6 One use of key frames is to define milestones throughout an animation, recording the state scene graph objects should be in at that point.

The code for the `masterTimeline` variable of the `RainDrop` class is conveniently reproduced next. It deals with the outermost part of the raindrop animation, firing off the ripples at regular beats.

```
masterTimeline = Timeline {
    keyFrames:
        for(i in [0..<numRipples]) KeyFrame {
            time: i*rippleGap;
            action: function() {
                ripples[i].rippleTimeline
                    .playFromStart();
            }
        }
};
```

In the example snippet we see only the first form of timeline in play. The `masterTimeline` is a `Timeline` object containing several `KeyFrame` objects, one for each ripple in the animation. Each key frame consists of two parts: the action to be performed (the `action`) and the point on the timeline when it should start (the `time`). The result is a timeline that works through the `ripples` sequence one by one, with a delay of `rippleGap` milliseconds between each, calling `playFromStart()` on the timeline inside each `Ripple` object and thereby starting its animation. In a nutshell, `masterTimeline` controls the triggering of each ripple in the raindrop; figure 5.7 shows how this works in diagrammatic form.

As the master timeline runs (shown vertically in figure 5.7), it triggers the individual ripple's animation (shown horizontally), which uses the second form of timeline to manipulate a circle over time. In the next section we'll take a look at this second, transitional timeline form.

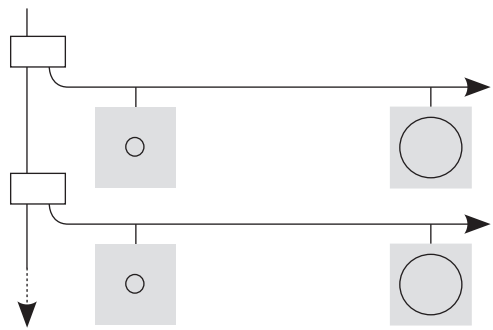

Figure 5.7 The master timeline awakes at regular intervals and fires off the next ripple's timeline. The effect is a raindrop of several ripples with staggered start times.

5.2.4 Interpolating variables across a timeline (at, tween, =>)

We've seen how `Timeline` and `KeyFrame` objects can be combined to call a piece of code at given points through the duration of an animation. This is like constructing a

timeline digitally, with actions triggered at set points along the course of the animation. But what happens if we wish to smoothly progress from one state to another?

The ripples in our animation demonstrate two forms of smooth animation: they grow outward toward their maximum radius, and they become progressively fainter as the animation runs. To do this we need a different type of key frame, one that marks waypoints along a journey of transition.

```
def rippleTimeline = Timeline {
    keyFrames: [
        at (0ms) {
            radius => 0;
            opacity => 1.0;
            strokeWidth => 10.0;
            visible => true;
        } ,
        at (1.5s) {
            radius => animRadius
                tween Interpolator.EASEOUT;
            opacity => 0.0
                tween Interpolator.EASEOUT;
            strokeWidth => 5.0
                tween Interpolator.LINEAR;
            visible => false;
        }
    ]
};
```

I've reproduced the `Timeline` constructed for the `Ripple` class—it uses a very unusual syntax compared to the one we saw previously in the `RainDrop` class. You may recall that earlier in this book I noted that one small part of the JavaFX Script syntax was to be explained later. Now it's time to learn all about that missing bit of syntax.

The `at/tween` syntax is a shortcut to make writing `Timeline` objects easier. In effect, it's a literal syntax for `KeyFrame` objects. Each `at` block contains the state of variables at a given point in the timeline, using a `=>` symbol to match value with variable. The duration literal following the `at` keyword is the point on the timeline to which those values will apply. Remember, those assignments will be made at some point in the future, when the timeline is executed—they do not take immediate effect.

Taking the previous example, we can see that at 0 milliseconds the `Ripple`'s `visible` attribute is set to `true`, while at 1.5 seconds (1500 milliseconds) it's set to `false`. Because invisible nodes are ignored when redrawing the scene graph, this shows the ripple at the start of the animation and hides it at the end. We also see changes to the ripple's `radius` (from 0 to `animRadius`, making the ripple grow to its desired size), its opacity (from fully opaque to totally transparent), and its line thickness (from 10 pixels to 5). But what about that `tween` syntax at the end of those lines?

The `tween` syntax tells JavaFX to perform a progressive *analog* change, rather than a sudden *digital* change. If not for `tween`, the ripple circle would jump immediately from 0 to maximum radius, fully opaque to totally transparent, and thick line to thin

line, once the 1.5 second mark was reached. Tweening makes the animation run through all the stages in between.

But you'll note we do more than just move in a linear fashion from one key frame to another; we actually define how the progression happens. The constants that follow the `tween` keyword (like `Interpolator.EASEOUT` and `Interpolator.LINEAR` in the example code) define the pace of transition across that part of the animation. In our example the ease-out interpolator starts slowly and builds up speed (a kind of soft acceleration), while a linear one maintains a constant speed across the transition (no wind up or wind down).

> **Limitations with the literal syntax, pre-1.2**
>
> It would seem that while the `at` syntax is happy to accept literal durations for times, the JavaFX Script 1.1 compiler had problems with variables. This made it difficult to vary the time of a key frame using a variable. The problem seems to have been addressed in the 1.2 compiler, but if you find yourself maintaining any old code, you need to be aware of this issue. In version 2 of this project you'll see a slightly more verbose syntax for key frames that has the same effect as `at/tween` but without this issue.

5.2.5 How the RainDrop class works

Before we move on to consider the bootstrap that will display our lovely new `RainDrop` class, I want to recap, step by step, exactly how the `RainDrop` works. We've covered quite a bit of new material in the last few pages, and it's important that you understand how it all fits together.

The `RainDrop` class is a `Node` that can be rendered in a JavaFX scene graph. It's constructed from other nodes, specifically several instances of the `Ripple` class, each of which draws and animates one circle (a ripple) in the drop animation. When the `RainDrop.start(x:Integer,y:Integer)` function is called, it fires up a `Timeline`, which periodically starts the timeline inside each `Ripple`, transitioning the radius, opacity, and stroke width of the circle to make it animate.

Now that we have something to animate, we need to plug it into a framework to show it onscreen. In the next section we'll see how that looks.

5.2.6 The LightShow class, version 1: a stage for our scene graph

To get our raindrops onscreen we need to create a scene graph window and hook the `RainDrop` class into its stage. A single raindrop wouldn't look very good, so how about we create multiple drops, which fire repeatedly as we move the mouse around? Listing 5.2 does just that!

Listing 5.2 LightShow.fx (version 1)

```
package jfxia.chapter5;

import javafx.scene.Scene;
import javafx.scene.input.MouseEvent;
```

```
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.stage.Stage;

import java.lang.System;

Stage {
    scene: Scene {
        def numDrops = 10;
        var currentDrop = 0;
        var lastDropTime:Long=0;
        var drops:RainDrop[];

        content: [
            Rectangle {
                width:400;  height:400;
                fill: Color.LIGHTCYAN;
                onMouseMoved: function(ev:MouseEvent) {
                    def t:Long =
                        System.currentTimeMillis();
                    if((t-lastDropTime) > 200) {
                        drops[currentDrop]
                            .start(ev.x,ev.y);
                        currentDrop =
                            (currentDrop+1) mod numDrops;
                        lastDropTime=t;
                    }
                }
            } ,
            drops = for(i in [0..<numDrops]) {
                RainDrop {};
            }
        ]
        width: 400;
        height: 400;
    }
    title: "LightShow v1";
    resizable: false;
}
```

Local variables, including RainDrop sequence

The background rectangle

Mouse event handler

Stage dimensions

Window configuration

We first encountered the `javafx.stage.Stage` class in the previous chapter. Its `scene` variable is the socket into which our scene graph should be plugged, which we do using the `javafx.scene.Scene` class.

The animation works by creating several instances of the `RainDrop` class, cycling through them over and over as the mouse moves. What happens if we run out of raindrops? Well, if we time things right, a `RainDrop` will have finished its animation by the time it is called into service again. By knowing how long each `RainDrop` animation takes and how frequently it is triggered, we can work out how many `RainDrop` nodes we need to continually run the animation.

Inside our `Scene` we first define some variables, local to that node:

- The `numDrops` variable defines the size of the `RainDrop` sequence—how many will be included in the scene graph, while `currentDrop` remembers which drop in the sequence is next to be animated.

- The `lastDropTime` variable records the time of the last drop animation, to ensure a reasonable gap between raindrops.
- `drops` is the `RainDrop` sequence itself.

We will manipulate these variables in an event handler. The first node in the `Scene` is a `Rectangle`, which is another kind of geometric shape similar to the `Circle` class. It will give our raindrop animation a colorful backdrop. We set the dimensions of the `Rectangle` within its parent and define a fill color. Then we assign an event handler to it.

```
onMouseMoved: function(ev:MouseEvent) {
    def t:Long = System.currentTimeMillis();
    if((t-lastDropTime) > 200) {
        drops[currentDrop].start(ev.x,ev.y);
        currentDrop = (currentDrop+1) mod numDrops;
        lastDropTime=t;
    }
}
```

The `onMouseMoved` variable is a function type, allowing us to attach event handling code to the `Rectangle` that responds to mouse movements across its surface. The example code has fewer line breaks, for extra readability. It works the same way as the button event handlers we saw in the Swing project in the last chapter, except it responds to mouse movement rather than button clicks. This event-handling code is the hub of the `LightShow` class; it's here that we initiate the raindrop animation. The code is assigned to the `Rectangle` because it covers the whole of the window interior, and thus it will receive movement events wherever the mouse travels.

But how does the event handler code work?

The first thing we do is get the current time from the computer's internal clock, using the Java method inside `java.lang.System`. We don't want to fire off new raindrops too quickly, so the next line is a check to see when we last started a fresh raindrop animation; if it's within the last 200 milliseconds, we exit without further action.

Assuming we're outside the time limit, we proceed by creating a fresh raindrop animation. This requires three steps: first we call `start(x:Integer,y:Integer)` on the next available raindrop, passing in the mouse event's x and y position, causing the class to begin animating around those coordinates. Then we move the `currentDrop` variable on to the next `RainDrop` in the sequence, wrapping around to the start if necessary, cycling through `RainDrop` objects as the mouse events are acted upon. Finally we store the current time, ready for the next handler invocation.

The `Rectangle` needs to access the `RainDrop` sequence from its `onMouseMoved` event handler, which is why we created a reference to the sequence as a local variable called `drops` before we declaratively created the `Rectangle`. Having added the `Rectangle` to the `Group` we can then add `drops`, so they'll be drawn above the `Rectangle`. Because JavaFX Script is an expression language, the assignment to the `drops` variable also acts as an assignment into the enclosing scene graph sequence.

5.2.7 Running version 1

Running the code is as simple as compiling both classes and starting up `jfxia.chapter5.LightShow` and waving your mouse over the window that appears. The effect is of circular patterns tracing the flow of your mouse, expanding and fading as they go. While they may serve no useful purpose, the application's visuals are (I hope) interesting and fun to play with. They demonstrate how rich animation can be created quite quickly from within JavaFX, with reasonably little code.

So far we've thrown a few shapes on screen and looked at how the scene graph groups things together. But the `RainDrop` is perhaps not the most efficient example of writing custom scene graph nodes. For a start, what happens if our custom node isn't a convenient subclass of an existing node? In the next section we'll see how using JavaFX's purpose-made `CustomNode` class helps us create unique custom nodes. We'll also be spinning a few psychedelic shapes on screen, so if you have a lava lamp around, now would be a good time to switch it on.

5.3 Total transformation: LightShow, version 2

Subclassing `CustomNode` is the recommended way of creating custom-made nodes in JavaFX. Although we managed perfectly well by subclassing `Group` for the `RainDrop`, a `CustomNode` subclass allows us a bit more control. For a start it includes a `create()` function that gets called when the node is created, acting as a lightweight constructor.

In our second version of the `LightShow` project we're going to write a `CustomNode` that will slot into the main bootstrap class, just like the `RainDrop`. We're also going to explore exotic uses of animation timelines, to bring a splash of Technicolor to our software.

5.3.1 The swirling lines animation

We'll start with a new class to create animated swirling lines, radiating out from a center point. The lines are like spokes on a wheel, spaced evenly around all 360 degrees of a ring, as in figure 5.8.

The `SwirlingLines` class uses transformations on `Rectangle` shapes to rotate and position each line within the scene graph. All the shapes in a given ring are contained within (and controlled via) a parent group, which is in turn linked to a custom node.

As with the `RainDrop`, we'll use the class multiple times in the `LightShow`, the end effect being several nested rings of colored lines rotating in different directions, while transitioning through several hues.

Figure 5.8 The `SwirlingLines` class creates a single ring of spokes, rotating around a central origin. Instances demonstrating different attribute settings are displayed.

Figure 5.8 shows the result. The animation will run continuously and will not interact with the user.

We have a lot of interesting new ideas to cover; all we need now is source code.

5.3.2 The SwirlingLines class: rectangles, rotations, and transformations

The `SwirlingLines` source code is presented in listing 5.3. It contains quite a host of instance variables for configuring its operation, from line length and thickness to the speed and direction of rotation. This will give us plenty of stuff to play with when we incorporate it into our project application a little later on.

Previously I mentioned that the lines in the finished application will continually change color. This class does not concern itself with the color changes, but it *does* bind a handy-dandy color variable, which some other class (the `LightShow` being a prime suspect) might want to manipulate. Listing 5.3 is the code.

Listing 5.3 SwirlingLines.fx

```
package jfxia.chapter5;

import javafx.animation.Interpolator;
import javafx.animation.KeyFrame;
import javafx.animation.Timeline;
import javafx.lang.Duration;
import javafx.scene.CustomNode;
import javafx.scene.Group;
import javafx.scene.Node;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.scene.transform.Transform;

package class SwirlingLines extends CustomNode {
    public-init var antiClockwise:Boolean = false;
    public-init var baseAngle:Number = 0.0;
    public-init var numLines:Integer = 12;
    public-init var rotateDuration:Duration = 1s;
    public-init var lineLength:Number = 100.0;
    public-init var lineThickness:Number = 20.0;
    public-init var centerRadius:Number = 20.0;
    package var color:Color;

    var rotateSlice:Number =
        (if(antiClockwise) -360.0 else 360.0) / numLines;
    var animRotateInc:Number;

    override function create():Node {
        def node = Group {
            content: for(i in [0..<numLines]) {
                Rectangle {
                    width: lineLength;
                    height: lineThickness;
                    fill: bind color;
                    transforms: [
                        Transform.rotate (
                            baseAngle + rotateSlice*i ,
```

External attributes

Internal attributes, mainly animation

Bound to external attributes

Transformation ops array

```
                        0,0
                    ) ,
                    Transform.translate (
                        centerRadius ,
                        0-lineThickness/2
                    )
                ];
            };
        };
        rotate: bind baseAngle + animRotateInc;
    };

    Timeline {
        repeatCount: Timeline.INDEFINITE;
        keyFrames: [
            KeyFrame {
                time: 0s;
                values: [
                    animRotateInc => 0.0
                ];
            } ,
            KeyFrame {
                time: rotateDuration;
                values: [
                    animRotateInc => rotateSlice
                        tween Interpolator.LINEAR
                ];
            }
        ];
    }.play();

    return node;
  }
}
```

Transformation ops array

Rotate Group

Run forever

Start of rotation

End of rotation

Start animation

The `SwirlingLines` class in listing 5.3 is a custom node, as denoted by its subclassing of the `javafx.scene.CustomNode` class. While subclassing `Group` was fine for our simple ripples, we need something more powerful for the effect we're building. `CustomNode` subclasses permit the building of complex multishape graphs inside their `create()` function, beyond just extending an existing predefined shape.

What's interesting about `create()` is that it gets called once, *before* the `init` or `postinit` blocks are run. Yes, you read that correctly: `create()` is run after the declared variables are set but before the class is fully initialized! Be sure to keep this in mind when designing your own `CustomNode` subclasses. Figure 5.9 is a diagrammatic rendering of what the `SwirlingLines` code produces.

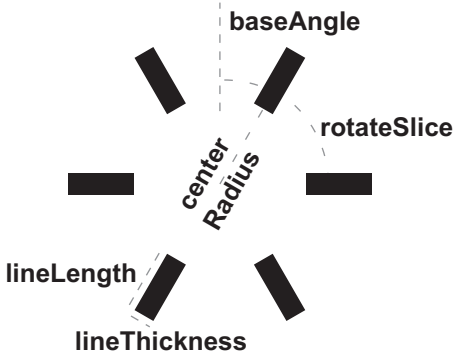

Figure 5.9 `SwirlingLines` creates a ring of rectangles, fully customizable from its instance variables.

Taking a look at the class we can clearly see several instance variables available for tailoring the class declaratively. Figure 5.9 shows the main initialization variables in diagram form, but here's a description of what they do:

- The `color` variable controls the line color (obviously!), while `lineLength` and `lineThickness` control the size of the lines. Since we'll be manipulating `color` throughout the run of the application, it has been made package visible, and its reference in the scene graph is bound.
- The `antiClockwise` flag determines the direction of animation, while `baseAngle` controls how the lines are initially oriented (the first line doesn't have to start at 0 degrees).
- The `numLines` determines how many lines form the ring; they will be evenly spaced around the total 360 degrees.
- The `rotateDuration` attribute controls how long it takes for the ring to perform one animation cycle. If there are 16 lines, this will be the time it takes to animate through one-sixteenth of a total 360-degree revolution.
- `centerRadius` details the empty space between the rotation center and the inner end of each line—in other words, how far away from the hub the lines are positioned.

There are some private variables, used to control the internal mechanics of the class.

- The `rotateSlice` value is the angle between each line in the ring. This is 360 divided by the number of lines. The value is used to constrain the animation (which we'll look at in a moment) and is either positive or negative, depending on in which direction the ring will spin.
- The `animRotateInc` object is the value we change during the timeline animation.

The rotation is performed at the group level, thanks to the group node's `rotate` variable being bound to `animRotateInc`. Although the ring will appear to rotate freely through a full 360 degrees, this is an optical illusion. The animation moves only between lines, so if there are four lines, our animation will continually run between just 0 and 90 degrees. (There's no great advantage in not making the ring spin a full 360 degrees; it just wasn't necessary to create the desired effect.)

The next part of the class is a function called `create()`, which returns a `Node`. The `CustomNode` class provides this function specifically for us to override with code to build our own scene graph structure (note the `override` keyword). The node we return from this function will be added to the scene graph, which is what we'll look at next.

5.3.3 Manipulating node rendering with transformations

The `create()` function is rather unusual, in that it gets called *before* the class's `init` block. This means if you're going to use `init` to set up any of your variables, they need to be bound in the scene graph to ensure the changes are reflected on screen.

Our `create()` function, in listing 5.3, does all the work of setting up the nodes in the swirling lines scene graph and defining the animation timeline. The first of these

Custom node initialization and older JavaFX versions

Be warned, JavaFX 1.2 was the first version of JavaFX in which `create()` was called before `init`. In previous versions `init` was called first, then `create()`, and finally `postinit`.

two responsibilities involves creating a sequence of `Rectangle` objects, the inner code for which is reproduced here with fewer line breaks:

```
Rectangle {
    width: lineLength;
    height: lineThickness;
    fill: bind color;
    transform: [
        Transform.rotate(baseAngle + rotateSlice*i, 0,0) ,
        Transform.translate(centerRadius , 0-lineThickness/2)
    ];
};
```

The function uses a `for` loop to populate a `Group` with `Rectangle` objects. Width, height, and fill color all reference the class variables, with `fill` being bound so it responds to changes. So far nothing new, but take a look at the `transform` assignment; what's going on there?

Transformations are discrete operations applied to a node (and thereby its children) during the process of rendering it to the screen. The `javafx.scene.transform.Transform` class contains a host of handy transformation functions that can be applied to our nodes. Transformations are executed in order, from first to last, and the order in which they are applied is often crucial to the result.

Let's consider the two operations in our example code: first we perform a rotation of a given angle around a given point, and then we move (translate) the node a given distance in the x and y directions. The first operation ensures the line is drawn at the correct angle, rotated around the origin (which is the center of the ring, recall). The second operation moves the line away from the origin but also centers it along its radial by moving *up* half its height. I realize it might be a little hard to visualize why this centering is necessary; figure 5.10 should clarify what's happening.

Without the negative y axis translation the rectangle would hang off the radial line like a flag on a flagpole. We want the rectangle to straddle the radial, and that's what the translation achieves.

As previously noted, it's important to consider the order in which transforming operations are performed. Turning 45 degrees and then walking forward five paces is not the same as walking forward

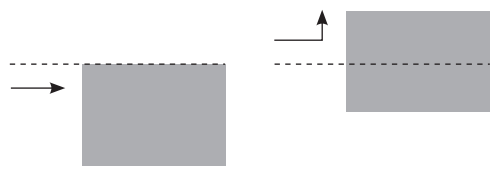

Figure 5.10 With and without centering: moving the `Rectangle` negatively in the y axis, by half its height, has the effect of centering it on its origin—in this case the radial spoke of a ring.

When *up* sometimes means *down*

When I say the second transformation moves the `Rectangle` "up," it's a relative term—up in relation to the rotation we previously applied. If thinking about this hurts your head, picture it this way: the screen is a giant wall, and tacked onto that wall are separate pieces of paper with drawings on them (like you'd find in an elementary school classroom, where students have their crayon masterpieces on display).

We can take any one of these pieces of paper and rotate in on the wall; for example, we could display a given picture upside-down. If we take one of the drawings down from the wall and erase an object, redrawing it higher up (like moving the sun farther up in the sky), then we tack the drawing back onto the wall upside-down, did we move the object up or down? In terms of the paper, we moved it up. But after we applied the rotation, we moved it down in terms of the overall wall. We have two coordinate spaces in operation here, the global one (the wall) and the local one (the paper).

To return to our project source code, when we said we moved the `Rectangle` "up," we meant in terms of its local coordinate space. That space (like the paper on the wall) is rotated, but it doesn't matter, because from a local point of view *up* still means *up* (*left* still means *left*, etc.), even if the rotation actually changes the effect in terms of the global space.

five paces and then turning 45 degrees; check out figure 5.11 if you don't believe me. Always remember the golden rule: each time a node is drawn, its transformations are applied in order, from first to last.

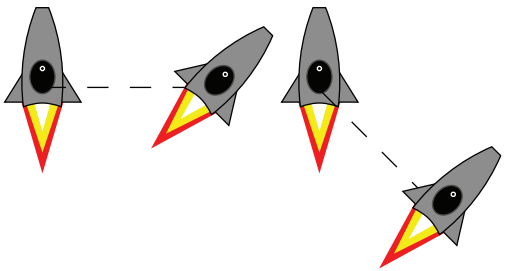

Figure 5.11 Two examples of transformations: translate and then rotate (left), and rotate and then translate (right). The order of the operations results in markedly different results.

Now that you understand transformations, let's look at the remainder of the code:

```
Timeline {
    repeatCount: Timeline.INDEFINITE;
    keyFrames: [
        KeyFrame {
            time: 0s;
            values: [
                animRotateInc => 0.0
            ];
        } ,
        KeyFrame {
            time: rotateDuration;
```

```
            values: [
                animRotateInc => rotateSlice
                    tween Interpolator.LINEAR
            ];
        }
    ];
}.play();
```

The example creates a timeline that runs continually once started, thanks to `Timeline.INDEFINITE`, with two `KeyFrame` objects, one marking its start and the other its end. All the timeline does is tween the variable `animRotateInc`, which the `Group` binds to. These changes cause the `Group`, and its `Rectangle` contents, to rotate.

The timeline looks a little different from the `at` syntax we saw in action in the `Ripple` class. There's no difference in terms of functionality; we're just writing out the `KeyFrame` objects longhand instead of using the briefer syntax. The `KeyFrame` code is quite easy to define declaratively; `time` is obviously the point at which the key frame should be active, while `values` is a comma-separated list (a sequence) of *attribute* => *value* definitions.

> **Limitations with the literal syntax, pre-1.2, part 2**
>
> As previously mentioned, the `at` syntax for describing key frames worked fine with time literals, but not variables, when used with JavaFX 1.1 compiler. However, the verbose syntax in our example doesn't seem to suffer from this problem. (As already noted, the issue seems to have been addressed in the 1.2 compiler.)

Once we've created the timeline, we kick it off immediately by calling its `play()` function. The `Timeline` class has a number of functions for controlling playback. The two we've seen in this project are `playFromStart()` and `play()`. The former restarts a timeline from the beginning, while the latter picks up where it left off. Because our timeline will run indefinitely, we can use either.

5.3.4 The LightShow class, version 2: color animations

Now that we have a swirling lines class, let's add it into our application class, `LightShow`. We also want to create some kind of psychedelic color effect as well. Listing 5.4 shows how the updates change the code.

Listing 5.4 LightShow.fx (version 2)

```
package jfxia.chapter5;

import javafx.animation.Interpolator;
import javafx.animation.KeyFrame;
import javafx.animation.Timeline;
import javafx.lang.Duration;
import javafx.scene.Scene;
import javafx.scene.input.MouseEvent;
```

```
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.stage.Stage;

import java.lang.System;

def sourceCols = [
     "#ff3333","#ffff33","#33ff33",
     "#33ffff","#3333ff","#ff33ff"
];
def colorShifts = for(i in [0..<6]) {
    ColorShifter {
        sourceColors: sourceCols;
        duration: 3s;
        offset: i;
    };
}

def sceneWidth:Number = 400;
def sceneHeight:Number = 400;
Stage {
    scene: Scene {
        def numDrops = 10;
        var currentDrop = 0;
        var lastDropTime:Long=0;
        def drop = for(i in [0..<numDrops]) {
            RainDrop {};
        }

        def lines = for(i in [0..<6]) {
            SwirlingLines {
                def ii = i+1;
                layoutX: sceneWidth/2;
                layoutY: sceneHeight/2;
                numLines: 6+i;
                color: bind colorShifts[i].color;
                centerRadius: (ii)*20;
                lineLength: (ii)*10;
                lineThickness: (ii)*3;
                antiClockwise: ((i mod 2)==0);
                rotateDuration: 1s/(ii);
            }
        };

        def rect = Rectangle {
            width: sceneWidth;
            height: sceneHeight;
            fill: Color.BLACK;
            onMouseMoved: function(ev:MouseEvent) {
                def t:Long =
                    System.currentTimeMillis();
                if((t-lastDropTime) > 200) {
                    drop[currentDrop].color =
                        colorShifts[0].color;
                    drop[currentDrop]
                        .start(ev.x,ev.y);
                    currentDrop =
```

Create six color animations

Swirling lines sequences, declaratively defined

Color shift our raindrops

```
                        (currentDrop+1) mod numDrops;
                    lastDropTime=t;
                }
            }
        };

        content: [
            rect ,
            lines ,
            drop
        ]
        width: sceneWidth;
        height: sceneHeight;
    }
    title: "LightShow v2";
    resizable: false;
    onClose: function() { FX.exit(); }
}

class ColorShifter {
    public-init var duration:Duration = 3s;
    public-init var sourceColors:String[];
    public-init var offset:Integer=0;

    var color:Color;
    var tLine:Timeline;

    init {
        def gap = duration / ((sizeof sourceColors)-1);
        Timeline {
            def arrSz = sizeof sourceColors;
            repeatCount: Timeline.INDEFINITE;
            keyFrames: for(i in [0..<arrSz+1]) {
                KeyFrame {
                    def pos = (offset+i) mod arrSz;
                    time: gap * i;
                    values: color =>
                        Color.web(sourceColors[pos])
                        tween Interpolator.LINEAR;
                }
            }
        }.play();
    }
}
```

Lines added to scene graph

Explicitly close window

Declaration variables

Output color

A KeyFrame for each color, wrapped around

We'll have a look at the color shifter first. At the start of listing 5.4 there's a sequence of colors called `sourceCols`, using web-style definitions (#rrggbb, as hex). Following that, a `for` loop creates seven `ColorShifter` objects.

The `ColorShifter` provides us with ever-shifting color, cycling through a collection of shades over a period of time. You can find its code at the bottom of listing 5.4. The external interface is as follows:

- The `duration` attribute is the time it will take to do one *full circle* of the colors.
- The sequence `sourceColors` provides the hues to cycle through, and `offset` is the index to use as the first color.
- `color` is the output—the current hue.

The class creates a timeline with a `KeyFrame` for each color, using tweening to ensure a smooth transition between each. The first color is used twice, at both ends of the animation, to ensure a smooth transition when wrapping around from last to first color. That's why the loop runs for one greater than the actual size of the source sequence and the `mod` operator is applied to the loop index to keep it within range. Once created, the `ColorShifter` timeline is started and runs continually.

That's it for the color animations. Returning to the scene graph, let's have a look at how the `SwirlingLines` are added to our `Group` node:

```
def lines = for(i in [0..<6]) {
    SwirlingLines {
        def ii = i+1;
        layoutX: fWidth/2;
        layoutY: fHeight/2;
        numLines: 6+i;
        color: bind colorShifts[i].color;
        centerRadius: (ii)*20;
        lineLength: (ii)*10;
        lineThickness: (ii)*3;
        antiClockwise: ((i mod 2)==0);
        rotateDuration: 1s/(ii);
    }
};
```

Nothing particularly unusual here. We create seven rings of lines, and each is bound to a different ever-changing `ColorShifter`. Because the `ColorShifter` objects were all declared with different source colors (we used a different `offset` each time), each ring pulses with a different part of the `sourceCols` input sequence. Each successive ring has more lines, longer and thicker, with a larger gap at the center. The rings alternate clockwise and counterclockwise, with outer rings rotating faster than inner ones.

Now that we have the code, let's see what happens when we run it.

5.3.5 Running version 2

Version 2 adds rotating patterns of lines and ever-changing colors, all animating merrily away without our having to get involved in any ugly code to draw them on screen as we'd need to if this was a Swing application using immediate mode rendering. Figure 5.12 shows the application running.

When you run the code, you'll see I added a few extra changes to version 1: the background is now black, and the raindrops are bound to a `ColorShifter` too. The effect is an explosion of color patterns across the window as the mouse is moved.

Figure 5.12 Version 2 of the project application, featuring both swirling lines and raindrops

5.4 Lost in translation? Positioning nodes in the scene graph

Before we round out this chapter, there's one topic we really *should* review: the relationship between a scene graph node's layout (`layoutX` and `layoutY`), its translation (`translateX` and `translateY`), and its coordinates. It's vital that you understand how these three work with one another.

Each scene graph node has a way of specifying its location within its own local space; for example, `Rectangle` has `x` and `y`, `Circle` has `centerX` and `centerY`, and so on. These define points within the node's own local coordinate space, quite separate to (although ultimately combined with) its location as a whole (see figure 5.13).

To borrow section 5.3.3's paper/wall analogy, if each shape is a drawing on a piece of paper tacked onto a wall, we might say `Rectangle.x` and `Rectangle.y` represent the rectangle's position within the paper, while `Rectangle.layoutX` and `Rectangle.layoutY` represent the paper's position within the wall. The former is the shape's location within its own space; the latter is its location within its parent's space. The important point to remember is this: the node's local space is unchanged no matter how it is rotated, bent, folded, or otherwise manipulated in the scene graph outside.

Actually, that isn't the whole story, because a shape's location isn't just determined by `layoutX` and `layoutY`. If you check the documentation for `Node`, you'll see there's a second set of coordinates, called `translateX` and `translateY`.

In versions of JavaFX prior to 1.2 the layout variables didn't exist, and the translate variables were used to position nodes. But 1.2 introduced the *controls* API, and with it more sophisticated layout management. The designers of JavaFX realized they needed to separate a node's layout position from any movement it might subsequently make as part of an animation; thus the concept of separate *layout* and *translation* coordinates was born. So, the location of a node is determined by its layout coordinates (its home within its parent), combined with its translation coordinates (where an animation has since moved it), combined with its own local coordinates after transformations have been applied.

Whew! Hopefully that clears everything up.

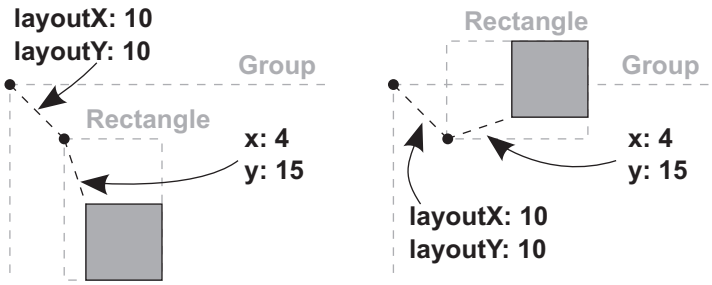

Figure 5.13 A `Rectangle` with its own local coordinates, translated within a `Group`. The local coordinates are not affected when the node is transformed in its parent's space, like a rotation by 270 degrees.

5.5 Bonus: creating hypertext-style links

We ended the last chapter with a bonus example listing, to frame the skills learned in more of a business/e-commerce context. This chapter's light synthesizer served as a fun way to introduce the scene graph, but it has little practical value beyond simply being entertaining. So before I sum up, let's indulge in another little detour, by way of listing 5.5.

Listing 5.5 Link.fx

```
import javafx.scene.*;
import javafx.scene.input.MouseEvent;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.scene.text.*;
import javafx.stage.Stage;

var border:Number = 20;
var hyperLink:Group = Group {
    var t:Text;
    var r:Rectangle;
    layoutX: border;  layoutY: border;
    content: [
        r = Rectangle {
            width: bind t.layoutBounds.width;
            height: bind t.layoutBounds.height;
            opacity: 0;
            onMouseClicked: function(ev:MouseEvent) {
                println("Ouch!");
            }
        } ,
        t = Text {
            content: "Link text";
            font: Font.font("Helvetica",56);
            textOrigin: TextOrigin.TOP;
            underline: bind r.hover;
            fill: bind if(r.pressed) Color.RED
                else Color.BLACK;
        }
    ]
}

Stage {
    scene: Scene {
        content: hyperLink;
        width: hyperLink.layoutBounds.width + border*2;
        height: hyperLink.layoutBounds.height + border*2;
    }
    resizable: false;
}
```

What the listing does is to create a hypertext-like button from some text. Actually, JavaFX 1.2 came with a link control as standard, but that doesn't mean we can't learn something from crafting our own. Figure 5.14 shows how it looks when running. The

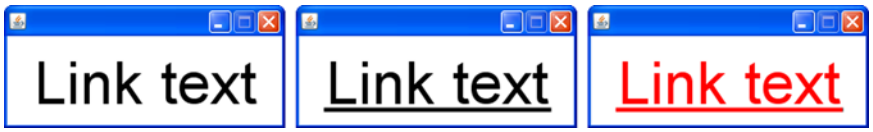

Figure 5.14 The link text depicted in three states: idle (left), underlined when hovered over (center), and red upon a mouse button press (right)

text is normally displayed black and undecorated, but it becomes underlined when the mouse is over it. When the mouse button is pressed, the color changes to red, before returning to black upon release.

The core of the code is in the `Rectangle` and `Text` objects used to populate the scene (via a `Group`). The `Text` object is akin to Swing's `JLabel`; it's used to include text within a scene graph. We'll cover `Text` in far more details next chapter, so for now please forgive me if I gloss over some of its details. The important point to note is that `Text`, like other scene graph nodes we encountered this chapter, behaves like a shape, meaning its concept of *area* is not limited to merely being rectangular. As the mouse travels over the text, it constantly enters and leaves the shape, the pointer passing inside and outside each letter.

Clearly we need the link to behave like a rectangle when it comes to its interaction with the mouse. To do this we employ an invisible `Rectangle` behind the `Text`, taking responsibility for mouse events. Note how the size of the `Rectangle` node is bound tightly to its companion `Text` node, while the `underline` decoration and `fill` color of the `Text` node are bound to the `Rectangle`'s `hover` state. Also note how the event function (which runs when the link is clicked) is attached to `onMouseClicked()` on the `Rectangle`, not the `Text` node.

Now that we have our own handmade hypertext link, we can customize it to our heart's content—perhaps make the underline fade in and out, or apply a drop shadow on mouse over. This was just an example to show how the scene graph sometimes requires a jot of lateral thinking when it comes to getting the effect you want. In the next chapter you'll see further examples of using nodes in clever ways, not just as proxies for event handling but to add padding around parts of the UI during layout and to define clipping areas to shape the visibility of a scene graph.

5.6 Summary

In this chapter we took our first look at the scene graph and played around with throwing shapes on screen. We saw multiple examples of timeline–based animation and explored how to define timelines to suit different purposes: triggering events at given moments and progressively transitioning variables between different states. We also witnessed how timelines could be used to animate more than just shapes onscreen. And to cap it all, we dabbled with mouse events.

The `LightShow` example isn't the most useful application in the world, and we didn't even wire it up to a sound source like true visualizations, but I hope you found

it suitably entertaining. You can use the `LightShow` application as a framework for plugging in and trying your own `CustomNode` experiments, if you wish. There are plenty of different transformations we didn't have space to cover in this chapter—you might want to try playing with some of them, distorting the `Rectangle` lines or even adding different shapes of your own and seeing what effects they create.

The bonus listing, I hope, got you thinking about how to adapt the techniques we used in this chapter for more practical purposes. Indeed in some ways it was a taste of what's to come, because in the very next chapter we'll be staying with the scene graph but looking at building a slightly more useful application (using video, no less!). We'll also be creating our own custom UI components and looking at some of the effects we can create using the scene graph.

For now, have fun extending and adapting the `LightShow`, and when you're ready I'll see you in the next chapter!