6 Moving pictures

This chapter covers

- Interacting with complex custom nodes
- Laying stuff out on screen, using containers
- Playing video
- Embedding fonts into an application

In previous chapters we developed an application using JavaFX's Swing wrappers and played around with the scene graph. Now it's time to turn our hands toward creating a UI that combines the functionality of a traditional desktop application with the visual pizzazz of the new scene graph library. And when it comes to desktop software, you don't often get showier than media players.

Some applications demand unique interfaces, games and media players being prime candidates. They don't just stick to the conventional form controls but create a less-rigid *experience*, with sophisticated use of imagery, sound, movement, and animation. In the chapter 5 we started to explore the JavaFX scene graph, and in the next chapter we'll be looking at JavaFX's standard form controls. So, by way of a bridge between the two, in this chapter we'll be getting far more interactive with the scene graph, by making it respond and animate to a range of mouse events. The nodes we'll develop will be far showier (and specialized to the UI experience of

this project) than the standard controls shipped with JavaFX or Swing. For this reason, we will be developing them as `CustomNode` classes, rather than formal `Control` classes. (There's no reason why they couldn't be written as full-blown controls, but I wanted this chapter to focus on getting more experience with the scene graph.)

We'll also look at using the media classes to play videos from the local hard disk. To try out the project (see figure 6.1) you'll need a directory of movie files, such as MPEG and WMV files.

The video player we'll be developing is very primitive, lacking much of the functionality of full-size professional players like Windows Media Player or QuickTime. A full player application would have been 10 times the size with little extra educational value. Although primitive in function, our player will require a couple of custom controls, each demonstrating a different technique or skill.

We'll also be adding a gradient fill to the background, and a real-time reflection effect will be applied to the video itself, making it look like it's resting on a shiny surface.

DOWNLOAD NEEDED

This project requires a few images, which can be downloaded along with the source code from the book's website: http://www.manning.com/JavaFXinAction.

Over the coming pages you'll learn about working with images and video, creating controls from scene graph shapes, and applying fancy color fills and effect. This is quite a busy project, with a lot of interesting ground to cover, so let's get started.

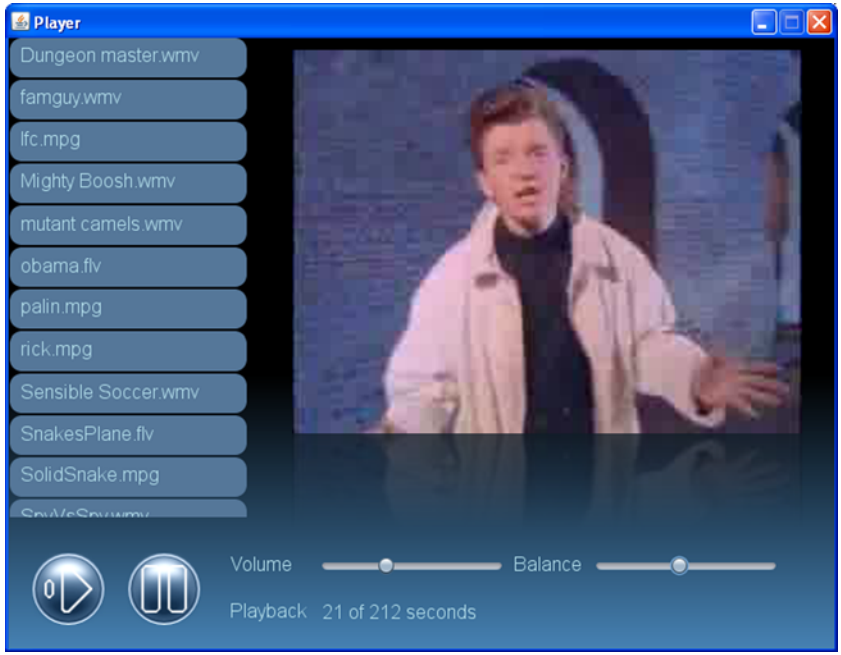

Figure 6.1 A preview of the simple video player we'll be building in this chapter

6.1 Taking control: Video Player, version 1

In this version of the player software we're focusing mainly on building the UI we'll need when we manipulate our video later on. Before JavaFX came along, getting video to work under Java would have deserved an entire book in itself. Thankfully, JavaFX makes things a lot easier. JavaFX's video classes are easy to use, so we don't have to devote the entire chapter to just getting video on screen.

You can see what this stage of the project looks like in figure 6.2.

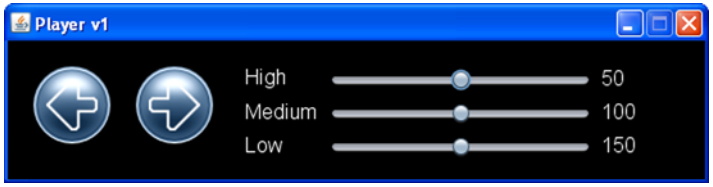

Figure 6.2
The interface for version 1 of our application

We'll begin simply enough, with the control panel that sits at the foot of the video player. It includes two examples of custom UI classes. The first is an image button, demonstrated to the left in figure 6.2; the second is a layout node, positioning the sliders and text to the right in figure 6.2.

We'll tackle the button first.

6.1.1 The Util class: creating image nodes

As before, the code is broken up into several classes and source files. But before we look at the button class itself, we'll take a short detour to consider a utility class. Quite often when we build software, the same jobs seem to keep coming up time and again, and sometimes it's useful to write small utility functions that can be called on by other parts of the code.

In our case the button we're writing needs to load images from a directory associated with the program—these are not images the user would choose but icons that come bundled with our player application. The code is shown in listing 6.1.

Listing 6.1 Util.fx

```
package jfxia.chapter6;

import javafx.scene.image.Image;

import java.io.File;
import java.net.URL;

package function appImage(f:String) : Image {
    Image {
        url: (new URL("{__DIR__}../../images/{f}")).toString();
    };
}
```

The script-level (static) function `appImage()` is used to load an application image from the project's images directory, such as icons, backgrounds, and other paraphernalia that

might constitute our UI. It accepts a string—the filename of the image to load—and returns a JavaFX `Image` class representing that image. The JavaFX `Image` class uses a URL as its source parameter, explaining the presence of the Java `URL` class. Have you noticed that strange symbol in the middle of the code: `__DIR__`? What does it do?

It's an example of a predefined variable for passing environment information into our running program.

- `__DIR__` returns the location of the current class file as a URL. It may point to a directory if the class is a `.class` bytecode file on the computer's hard disk, or it may point to a JAR file if the class has been packaged inside a Java archive.
- `__FILE__` returns the full filename and path of the current class file as a URL.
- `__PROFILE__` returns either `browser`, `desktop`, or `mobile`, depending on the environment the JavaFX application is running in.

Note how both `__FILE__` and `__DIR__` are URLs instead of files. If the executing class lives on a local drive, the URL will use the file: protocol. If the class was loaded from across the internet, it may take the form of an http:-based address.

Most of you should have realized the `appImage()` function isn't the most robust piece of code in the world. It relies on the fact that our classes live in a package called `jfxia.chapter6`, which resolves to two directories deep from the application's root directory. It backs out of those directories and looks for an images directory living directly off the application root. If we were to package the whole application into a JAR, this brittle assumption would break. But for now it's enough to get us going.

6.1.2 The Button class: scene graph images and user input

The `Button` class creates a simple push button of the type we saw in our Swing example in chapter 4. However, this one is built entirely using the scene graph and has a bit of animation magic when it's pressed. The `Button` is a very simple component, which offers an ideal introduction to creating sophisticated, interactive, custom nodes.

The button is constructed from two bitmap graphics: a background frame and a foreground icon. When the mouse moves over the button its color changes, and when clicked it changes again, requiring three versions of the background image: the *idle* mode image, the *hover* image, and the *pressed* (clicked) image. See figure 6.3.

When the button is pressed, the copy of the icon expands and fades, creating a pleasing ghosted zoom animation. Figure 6.3 demonstrates the effect: the arrow icon animates over the blue circle background. We'll be constructing the button from scratch, using a `CustomNode`, and implementing its animation as well as providing our own event handlers (because a button that stays mute when pressed is about as much use as the proverbial chocolate teapot).

Figure 6.3 The button is constructed from two bitmap images: a background (blue circle) and icon (arrow). When the button is pressed, a ghost of its icon expands and fades.

Enough theory. Let's look at the code in listing 6.2.

BROKEN LISTINGS

Because the button we're developing is a little more involved than the code we've seen thus far, I've broken its listing into two chunks, with accompanying text.

Listing 6.2 Button.fx (part 1)

```
package jfxia.chapter6;

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.input.MouseEvent;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.scene.shape.Rectangle;
import javafx.util.Math;

def backIdleIm:Image = Util.appImage("button_idle.png");
def backHoverIm:Image = Util.appImage("button_high.png");
def backPressIm:Image = Util.appImage("button_down.png");

package class Button extends CustomNode {
    public-init var iconFilename:String;
    public-init var clickAnimationScale:Number = 2.5;
    public-init var clickAnimationDuration:Duration = 0.25s;
    public-init var action:function(ev:MouseEvent);

    def foreImage:Image = Util.appImage(iconFilename);
    var backImage:Image = backIdleIm;
    def maxWidth:Number = Math.max (
        foreImage.width*clickAnimationScale ,
        backImage.width
    );
    def maxHeight:Number = Math.max (
        foreImage.height*clickAnimationScale ,
        backImage.height
    );

    var animButtonClick:Timeline;
    var animScale:Number = 1.0;
    var animAlpha:Number = 0.0;
    var iconVisible:Boolean = false;

    // ** Part 2 is listing 6.3
```

Button frame images

External interface variables

Private variables

Which is the bigger image?

Animation variables

Listing 6.2 covers the first part of our custom button, including the variable definitions. The first thing we see are script variables to use as the button background images. Because these are common to all instances of our `Button` class, we save a few bytes by loading them just once.

Our `Button` class extends `javafx.scene.CustomNode`, which is the recognized way to create new scene graph nodes from scratch. Inside the class itself we find several external interface variables, used to configure its behavior:

- The `iconFilename` variable holds the filename of the icon image.
- The `clickAnimationScale` and `clickAnimationDuration` variables control the size and timing of the fade/zoom effect.
- The `action` function type is for a button press event handler.

There are also internal implementation variables:

- The `foreImage` and `backImage` are the button's current background and foreground images.
- Variables `maxWidth` and `maxHeight` figure out how big the button should be, based on whichever is larger, the foreground or the background image.
- The variables `animButtonClick`, `animScale`, `animAlpha`, and `iconVisible` all form part of the fade/zoom animation.

For `CustomNode` classes the `create()` function is called before the `init` block is run. This means we need to initialize the `foreImage`, `backImage`, `maxWidth`, and `maxHeight` variables as part of their definition, so they're ready to use when `create()` is called.

Custom node initialization

It's worth repeating: starting with JavaFX 1.2, `create()` is called before `init` and `postinit`; don't get caught! Give all key variables default values as part of their definition. Do not initialize them in the `init` block.

A quick tip: remember that object variables are initialized in the order in which they are specified in the source file, so if you ever need to preload private variables, make sure any public variables they depend on are initialized first.

Listing 6.3, the second half of the code, is where we create our button's scene graph.

Listing 6.3 Button.fx (part 2)

```
// ** Part 1 in listing 6.2
override function create() : Node {
    def n = Group {
        layoutX: maxWidth/2;
        layoutY: maxHeight/2;
        content: [
            Rectangle {
                x: 0-(maxWidth/2);
                y: 0-(maxHeight/2);
                width: maxWidth;
                height: maxHeight;
                opacity: 0;
            } ,
            ImageView {
                image: bind backImage;
                x: 0-(backImage.width/2);
                y: 0-(backImage.height/2);
```

Force dimensions

Background image

```
            onMouseEntered: function(ev:MouseEvent) {
                backImage = backHoverIm;
            }
            onMouseExited: function(ev:MouseEvent) {
                backImage = backIdleIm;
            }
            onMousePressed: function(ev:MouseEvent) {
                backImage = backPressIm;
                animButtonClick.playFromStart();
                if(action!=null)  action(ev);
            }
            onMouseReleased: function(ev:MouseEvent) {
                backImage = backHoverIm;
            }
        } ,
        ImageView {
            image: foreImage;
            x: bind (0-foreImage.width)/2;
            y: bind (0-foreImage.height)/2;
            opacity: bind 1-animAlpha;
        } ,
        ImageView {
            image: foreImage;
            x: bind 0-(foreImage.width/2);
            y: bind 0-(foreImage.height/2);
            visible: bind iconVisible;
            scaleX: bind animScale;
            scaleY: bind animScale;
            opacity: bind animAlpha;
        }
    ]
};

animButtonClick = Timeline {
    keyFrames: [
        KeyFrame {
            time: 0s;
            values: [
                animScale => 1.0  ,
                animAlpha => 1.0 ,
                iconVisible => true
            ]
        } ,
        KeyFrame {
            time: clickAnimationDuration;
            values: [
                animScale => clickAnimationScale
                    tween Interpolator.EASEOUT ,
                animAlpha => 0.0
                    tween Interpolator.LINEAR ,
                iconVisible => false
            ]
        }
    ];
};
```

Mouse enters node

Mouse leaves node

Mouse button down

Mouse button up

Icon image

Animation image

Animation timeline

```
        return n;
    }
}
```

At the top of listing 6.3 is the `create()` function, where the scene graph for this node is constructed. This is a function inherited from `CustomNode`, which is why `override` is present, and it's the recognized place to build our graph.

As we've come to expect, the various elements are held in place with a `Group` node. Moving the button's x and y coordinate space (`layoutX` and `layoutY`) into the center makes it easier to align the elements of the button. The `Group` is constructed from one `Rectangle` and three `ImageView` objects (figure 6.4). The `Rectangle` is invisible and merely forces the dimensions of the button to the maximum required size to prevent resizing (and jiggling neighboring nodes around) during animations.

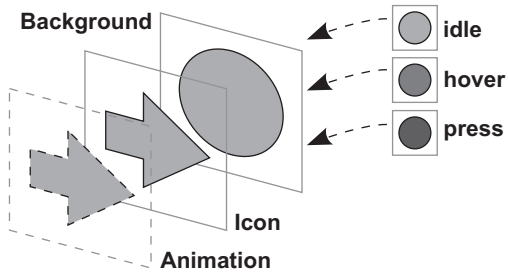

Figure 6.4 Ignoring the invisible `Rectangle` (used for sizing), there are three layers in our button.

In front of the rectangle there are three `ImageView` objects. What's an `ImageView`? It's yet another type of scene graph node. This one displays `Image` objects; the clue is in the class name. Our button requires three images (figure 6.4):

- The button background image, which changes when the mouse hovers over or clicks the button.
- The icon image, which displays the actual button symbol.
- The animation image, which is used in the fade/zoom effect when the button is pressed. This is a copy of the icon image, hidden when the animation isn't playing.

Look at the code for the first `ImageView` declaration. Like other `Node` subclasses, the `ImageView` can receive events and has a full complement of event-handling function types into which we can plug our own code. In the case of the background `ImageView`, we've wired up handlers to change its image when the mouse rolls into or out of the node and when the mouse button is pressed and released. The button press is by far the most interesting handler, as it not only changes the image but launches the fade/zoom animation and calls any `action` handler that might be linked to our `Button` class.

The second `ImageView` in the sequence displays the regular icon—the only cleverness is that it will fade into view as the animating fade/zoom icon fades out. Subtracting the current animation opacity from 1 means this image always has the opposite opacity to the animation icon image, so as the zooming icon fades out of view, the regular icon reappears, creating a pleasing full-circle effect.

The third `ImageView` in the sequence is the animation icon. It performs the fabled fade/zoom, and as you'd expect it's heavily bound to the object variables, which are manipulated by the animation's timeline.

And speaking of timelines, the `create()` function is rounded off by a classic start/finish key frame example, not unlike the examples we saw in chapter 5. The animation icon's size (scale) and opacity (alpha) are transitioned, while the animation `ImageView` is switched on at the start of the animation and off at the end. Simple stuff!

And so that, ladies and gentlemen, boys and girls, is our `Button` class. It's not perfect (indeed it has one minor limitation we'll consider later, when plugging it into our application), but it shows what can be achieved with only a modest amount of effort.

6.1.3 The GridBox class: lay out your nodes

Our button class is ready to use. Now it's time to turn our attention to the other piece of custom UI code in this stage of the application: the layout node. Figure 6.5 shows the effect we're after: the text and slider nodes in that screen shot are held in a loose grid, with variable-size columns and rows that adapt to the width or height of their contents.

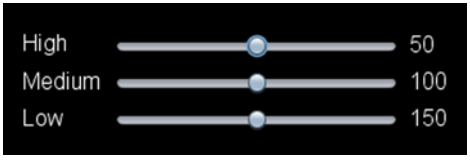

Figure 6.5 The `GridBox` node positions its children into a grid, with flexible column and row sizes.

This is not an effect we can easily construct with the standard layout classes in `javafx.scene.layout`. If you check the API documentation, you'll see there's a really handy `Tile` class that lays out its contents in a grid. But `Tile` likes to have uniform column and row sizes, and we want our columns and rows to size themselves individually around their largest element. So we have no option but to create our own layout node, and that's just what listing 6.4 does.

Listing 6.4 GridBox.fx

```
package jfxia.chapter6;

import javafx.geometry.HPos;
import javafx.geometry.VPos;
import javafx.scene.Node;
import javafx.scene.layout.Container;

package class GridBox extends Container {
    public-init var columns:Integer = 5;
    public-init var nodeHPos:HPos = HPos.LEFT;
    public-init var nodeVPos:VPos = VPos.TOP;
    public-init var horizontalGap:Number = 0.0;
    public-init var verticalGap:Number = 0.0;

    override function doLayout() : Void {
        def nodes = getManaged(content);

        def sz:Integer = sizeof nodes;
        var rows:Integer = (sz/columns);
        rows += if((sz mod columns) > 0) 1 else 0;
```

Width in columns

Alignment

Gap between nodes

Content to lay out

How many rows?

```
        var colSz:Number[] = for(i in [0..<columns]) 0.0;
        var rowSz:Number[] = for(i in [0..<rows]) 0.0;
        for(n in nodes) {
            def i:Integer = indexof n;
            def c:Integer = (i mod columns);
            def r:Integer = (i / columns).intValue();
            def w:Number = getNodePrefWidth(n);
            def h:Number = getNodePrefHeight(n);
            if(w > colSz[c])  colSz[c]=w;
            if(h > rowSz[r])  rowSz[r]=h;
        }

        var x:Number = 0;
        var y:Number = 0;
        for(n in nodes) {
            def i:Integer = indexof n;
            def c:Integer = (i mod columns);
            def r:Integer = (i / columns).intValue();

            layoutNode(n , x,y,colSz[c],rowSz[r] ,
                nodeHPos,nodeVPos);

            if(c < (columns-1)) {
                x+=(colSz[c] + horizontalGap);
            }
            else {
                x=0;  y+=(rowSz[r] + verticalGap);
            }
        }
    }
}
```

Find maximum col/row size

Position node

Next position

There are two ways to create custom layouts in JavaFX: one produces layout nodes that can be used time and time again, and the other is useful for case-specific one-shot layouts. In listing 6.4 we see the former (reusable) approach.

To create a layout node we subclass `Container`, a type of `Group` that understands layout. As well as proving a framework to manage node layout, `Container` has several useful utility functions we can use when writing node-positioning code.

Our `Container` subclass is called `GridBox`, and has these public variables:

- `columns` determines how many columns the grid should have. The number of rows is calculated based on this value and the number of managed (laid-out) nodes in the `content` sequence.
- `nodeHPos` and `nodeVPos` determine how nodes should be aligned within the space available to them when being laid out.
- `horizontalGap` and `verticalGap` control the pixel gap between rows and columns.

To make the code simpler, all the configuration variables are `public-init` so they cannot be modified externally once the object is created. The `doLayout()` function is overridden to provide the actual layout code. Code inherited from the `Container` class will call `doLayout()` whenever JavaFX thinks a layout refresh is necessary.

To perform the actual layout, first we scan our child nodes to work out the width of each column and the height of each row. Rather than read the `content` sequence directly, we use a handy function inherited from `Container`, `getManaged()`, to process only those nodes that require layout. Two more inherited functions, `getNodePrefWidth()` and `getNodePrefHeight()`, extract the preferred size from each node. Having worked out the necessary column/row sizes, we do a second pass to position each node. Yet another inherited function, `layoutNode()`, positions the node within a given area (x, y, width and height) using the node's own layout preferences (if specified) or `nodeHPos` and `nodeVPos`. The result is the flexible grid we want, with each node appropriately aligned within its cell.

Layout using the object literal syntax

The `GridBox` class is an example of a layout node: a fully fledged subclass of `Container` that can be used over and over. But what if we want a specific, one-time-only layout; do we need to create a subclass every time?

The `Container` class has a sibling called `Panel`. It does the same job as `Container`, except its code can be plugged in via function types. We can drop a `Panel` into our scene graph with an anonymous function to provide the layout code, allowing us to create one-shot custom layouts without the use of a subclass. A full example of `Panel` in action will be presented in the next chapter.

Now we have a functioning grid layout node class; all we need are nodes to use it with, a problem we'll remedy next.

6.1.4 The Player class, version 1

We have our two custom classes, one a control node, the other a layout node. Before going any further we should give them a trial run in a prototype version of our video player. Listing 6.5 will do this for us.

Listing 6.5 Player.fx (version 1)

```
package jfxia.chapter6;

import javafx.geometry.VPos;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.control.Slider;
import javafx.scene.input.MouseEvent;
import javafx.scene.layout.LayoutInfo;
import javafx.scene.paint.Color;
import javafx.scene.text.Font;
import javafx.scene.text.Text;
import javafx.scene.text.TextOrigin;
import javafx.stage.Stage;

def font = Font { name: "Helvetica";  size: 16; };

Stage {
```

Handy font declaration

```
    scene: Scene {
        content: [
            Button {                                    // Left button
                iconFilename: "arrow_l.png";
                action: function(ev:MouseEvent) {
                    println("Back");
                };
            } ,
            Button {                                    // Right button (note the layoutX)
                layoutX: 80;
                iconFilename: "arrow_r.png";
                action: function(ev:MouseEvent) {
                    println("Fore");
                };
            } ,
            GridBox {                                   // Our GridBox in action
                layoutX: 185;  layoutY: 20;
                columns: 3;
                nodeVPos: VPos.CENTER;
                horizontalGap: 10;  verticalGap: 5;

                var max:Integer=100;
                content: for(l in ["High","Medium","Low"]) {   // Loop to add rows
                    var sl:Slider;
                    var contentArr = [
                        Text {                          // Left-hand label
                            content: l;
                            font: font;
                            fill: Color.WHITE;
                            textOrigin: TextOrigin.TOP;
                        } ,
                        sl = Slider {                   // The slider itself
                            layoutInfo:
                                LayoutInfo { width: 200; }
                            max: max;
                            value: max/2;
                        } ,
                        Text {                          // Bound display value
                            content: bind sl.value
                                .intValue().toString();
                            font: font;
                            fill: Color.WHITE;
                            textOrigin: TextOrigin.TOP;
                        }
                    ];
                    max+=100;
                    contentArr;                         // Add row to GridBox
                }
            }
        ];
        fill: Color.BLACK;
    };
    width: 550;  height: 140;
    title: "Player v1";
    resizable: false;
}
```

The code displays the two classes we developed: two image buttons are combined with JavaFX slider controls, using our grid layout.

I mentioned very briefly at the end of the section dealing with the `Button` class that our button code has a slight limitation, which we'd discuss later. Now is the time to reveal all. The click animation used in our `Button` class introduces a slight headache: the animation effect expands beyond the size of the button itself. Although it creates a cool zoom visual, it means padding is required around the perimeter, accommodating the effect when it occurs. This is the purpose of the transparent `Rectangle` that sits behind the other nodes in the `Button`'s internal scene graph. Without this padding the button would grow in size as the animation plays, which might cause its parent layout node to continually reevaluate its children, resulting in a jostling effect on screen as other nodes move to accommodate the button.

To solve this problem we need to absolutely position our buttons, overlapping them so they mask their oversized padding. And this is what listing 6.5 does, by using the `layoutX` variable.

Following the two buttons in the listing we find an example of our `GridBox` in use. Its content is formed using a `for` loop, adding three nodes (one whole row) with each pass. The first and last are `Text` nodes, while the middle is a `Slider` node. The `javafx.scene.text.Text` nodes simply display a string using a font, not unlike the `SwingLabel` class. However, because this is a scene graph node, it has a `fill` (body color) and a `stroke` (perimeter color), as well as other shape-like capabilities. By default a `Text` node's coordinate origin is measured from the font's baseline (the imaginary line on which the text rests, like the ruled lines on writing paper), but in our listing we relocate the origin to the more convenient top-left corner of the node.

The `Slider`, as its name suggests, allows the user to pick a value between a given minimum and a maximum by dragging a thumb along a track. We explicitly set the layout width of the control by assigning a `LayoutInfo` object. When our `GridBox` class uses `getNodePrefWidth()` and `getNodePrefHeight()` to query each node's size, this layout data is what's being consulted (if the `LayoutInfo` isn't set, the node's `getPrefWidth()` and `getPrefHeight()` functions are consulted.) The final `Text` node on each row is bound to the current value of this slider, and its text content will change when the associated slider is adjusted.

Version 1 of our application is complete!

6.1.5 Running version 1

Running version 1 gives us a basic control panel, as revealed by figure 6.6. Although the code is compact, the results are hardly crude. The buttons are fully functional,

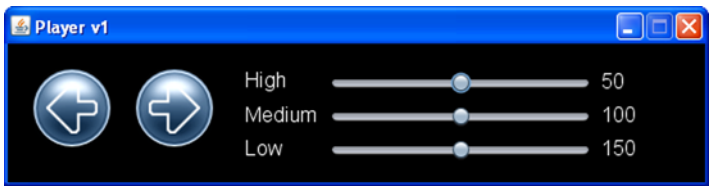

Figure 6.6 Our custom button and layout nodes on display

have their own event handler into which code can be plugged, and sport a really cool animation effect when pressed. The layout node makes building the slider UI much easier, yet it's still appropriately configurable.

I'll leave it up to you, the reader, to polish off our custom classes with your own bells and whistles. The `GridBox` in particular could become a really powerful layout class with not a great deal of extra work. The additional code wouldn't be of value from a demonstration viewpoint (that's why I didn't add it myself), but I encourage you to use version 1 as a test bed to try out your own enhancements.

So much for custom buttons. Did I hear someone ask when we will start playing with video? Good question. In the second, and final, part of this project we develop our most ambitious custom node yet—and, yes, finally we get to play with some video.

6.2 Making the list: Video Player, version 2

In this part of the chapter we have two objectives. The first is to incorporate a video playback node into our scene graph; the second is to develop a custom node for listing and choosing videos. Figure 6.7 shows what we're after.

Figure 6.7 shows the two new elements in action. The list allows the user to pick a video, and the video playback node will show it. Our control panel will interact with the video as it plays, pausing or restarting the action and adjusting the sound. The list display down the side will use tween-based animations, to control not only rollover effects but also its scrolling.

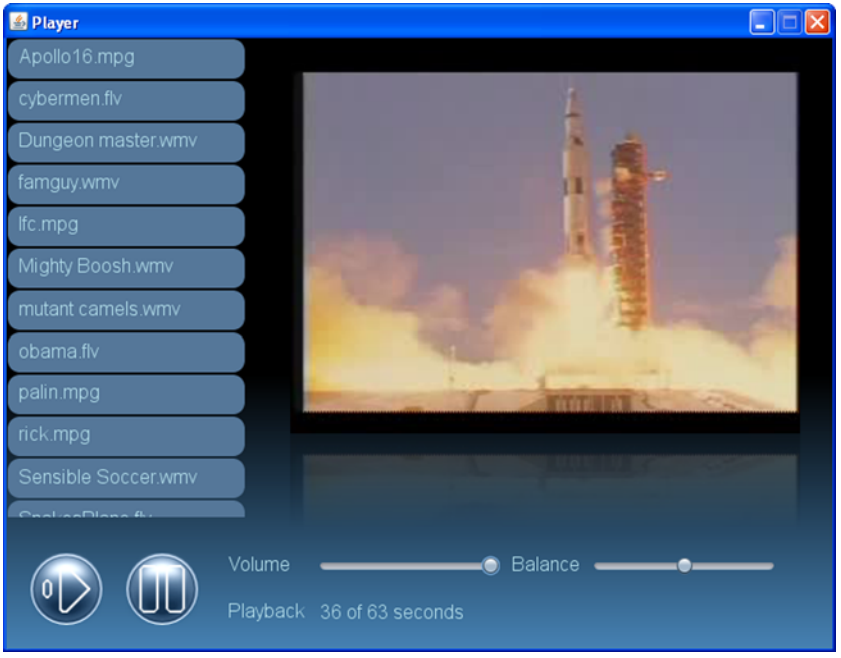

Figure 6.7 Lift off! Our control panel (bottom) is combined with a new list (left-hand side) and video node (center) to create the final player.

We need to develop this list node first, so that's where we'll head next.

6.2.1 The List class: a complex multipart custom node

The `List/ListPane` code is quite complex, indeed so complex that it's been broken into two classes. `List` is an interior node for displaying a list of strings and firing `action` events when they are clicked. `ListPane` is an outer container node that allows the `List` to be scrolled. In figure 6.8 you can see how the list parts fit together.

Rather than using a scrollbar, I thought we might attempt something a little different; the list will work in a vaguely iPhone-like fashion. A press and drag will scroll the list, with inertia when we let go, while a quick press and release will be treated as a click on a list item. We start with just the inner `List` node, which I've broken into two parts to avoid page flipping. The first part is listing 6.6.

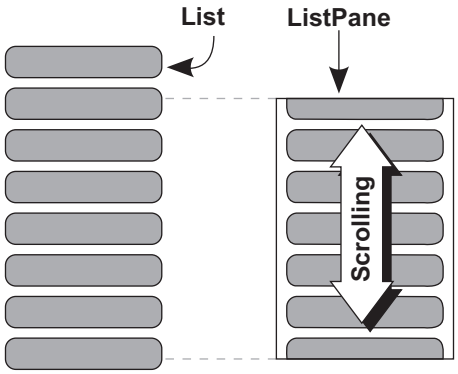

Figure 6.8 The `List` and `ListPane` classes will allow us to present a selection of movie files for the user to pick from.

Listing 6.6 List.fx (part 1)

```
package jfxia.chapter6;

import javafx.animation.Interpolator;
import javafx.animation.KeyFrame;
import javafx.animation.Timeline;
import javafx.scene.CustomNode;
import javafx.scene.Group;
import javafx.scene.Node;
import javafx.scene.input.MouseEvent;
import javafx.scene.layout.VBox;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.scene.text.Font;
import javafx.scene.text.Text;
import javafx.scene.text.TextOrigin;

package class List extends CustomNode {
    package var cellWidth:Number = 150;
    package var cellHeight:Number = 35;

    public-init var content:String[];
    public-init var font:Font = Font {};
    public-init var foreground:Color = Color.LIGHTBLUE;
    public-init var background:Color = Color.web("#557799");
    public-init var backgroundHover:Color = Color.web("#0044AA");
    public-init var backgroundPressed:Color = Color.web("#003377");
```

List item dimensions

```
public-init var action:function(n:Integer);

def border:Number = 1.0;
var totalHeight:Number;

override function create() : Node {
    VBox { content: build(); }
}
// ** Part 2 is listing 6.7
```

Create scene graph

At the head of the code is our usual collection of variables for controlling the class:

- `cellWidth` and `cellHeight` are the dimensions of the items on screen. They need to be manipulated by the `ListPane` class, so we've given them package visibility.
- `content` holds the strings that define the list labels.
- `font`, `foreground`, `background`, `backgroundHover`, and `backgroundPressed` control the font and colors of the list.
- The function type `action` is our callback function.
- `border` holds the gap around items in the list, and `totalHeight` stores the pixel height of the list. Both are private variables.

Looking at the `create()` code, we see a `VBox` being fed content by a function called `build()`. `VBox` is a layout node that stacks its contents one underneath the other—precisely the functionality we need. But what about the `build()` function, which creates its contents? Look at the next part of the code, in listing 6.7.

Listing 6.7 List.fx (part 2)

```
// ** Part 1 is in listing 6.6
function build() : Node[] {
    for(i in [0..<sizeof content]) {
        var t:Text;
        var r:Rectangle;
        def g = Group {
            content: [
                Rectangle {
                    width: bind cellWidth;
                    height: bind cellHeight;
                    opacity: 0.0;
                },
                r = Rectangle {
                    x:border;  y:border;
                    width: bind cellWidth-border*2;
                    height: bind cellHeight-border*2;
                    arcWidth:20;  arcHeight:20;
                    fill: background;
                    onMouseExited: function(ev:MouseEvent) {
                        anim(r,background);
                    };
                    onMouseEntered: function(ev:MouseEvent) {
                        r.fill = backgroundHover;
                    }
```

For each list item

Hidden sizing rectangle

List rectangle

```
                        onMousePressed: function(ev:MouseEvent) {
                            r.fill = backgroundPressed;
                        }
                        onMouseClicked: function(ev:MouseEvent) {
                            r.fill = backgroundHover;
                            if(action!=null) { action(i); }
                        }
                    } ,
                    t = Text {                                    <-- Label text
                        x: 10;  y: border;
                        content: bind content[i];
                        font: bind font;
                        fill: bind foreground;
                        textOrigin: TextOrigin.TOP;
                    }
                ];
            };
            t.y += (r.layoutBounds.height-                        <-- Center text vertically
                t.layoutBounds.height)/2;
            totalHeight += g.layoutBounds.height;
            g;
        }
    }

    function anim(r:Rectangle,c:Color) : Void {                   <-- Animate background
        Timeline {
            keyFrames: [
                KeyFrame {
                    time: 0.5s;
                    values: [
                        r.fill => c
                            tween Interpolator.LINEAR
                    ];
                }
            ]
        }.playFromStart();
    }
}
```

The `build()` function returns a sequence of nodes, each a `Group` consisting of two `Rectangle` nodes and a `Text` node. The first node enforces an empty border on all four sides of each item. The second `Rectangle` is the visible box for our item; it also houses all the mouse event logic. Finally, we have the label itself, as a `Text` node. For easy handling we use the top of the text as its coordinate origin, rather than its baseline.

Both the background `Rectangle` and `Text` are assigned to variables (since JavaFX Script is an expression language, this won't prevent them from being added to the `Group`). But why? Take a look at the code immediately after the `Group` declaration; using those variables we vertically center the `Text` within the `Rectangle`, and that's why we needed references to them.

Now let's consider the mouse handlers. Entering the item sets the background rectangle fill color to `backgroundHover`, while exiting the item kicks off a `Timeline`

(via the `anim()` function) to slowly return it to `background`. This slow fade creates a pleasing trail effect as the mouse moves across the list.

Pressing the mouse button sets the color to `backgroundPressed`, but we don't bother with the corresponding button release event; instead, we look for the higher-level clicked event, created when the user taps the button as opposed to a press and hold. The click event fires off our own `action` function, which can be assigned by outside code to respond to list selections.

The `List` class is only half of the list display; it's almost useless without its sibling, the `ListPane` class. That's where we're headed next.

6.2.2 The ListPane class: scrolling and clipping a scene graph

Now that we've seen the `List` node, let's consider the outer container that scrolls it. Check out listing 6.8.

Listing 6.8 ListPane.fx (part 1)

```
package jfxia.chapter6;

import javafx.animation.Interpolator;
import javafx.animation.KeyFrame;
import javafx.animation.Timeline;
import javafx.scene.CustomNode;
import javafx.scene.Group;
import javafx.scene.Node;
import javafx.scene.input.MouseEvent;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;

package class ListPane extends CustomNode {
    public-init var content:List;

    package var width:Number = 150.0
        on replace oldVal = newVal {
            if(content!=null)
                content.cellWidth = newVal;
        };
    package var height:Number = 300.0;
    package var scrollY:Number = 0.0
        on replace oldVal = newVal {
            if(content!=null)
                content.translateY = 0-newVal;
        };

    var clickY:Number;
    var scrollOrigin:Number;
    var buttonDown:Boolean = false;
    var dragDelta:Number;
    var dragTimeline:Timeline;
    var noScroll:Boolean =
        bind content.layoutBounds.height < this.height;
    // ** Part 2 is listing 6.9
```

Pass width on to List

Position List within pane

Drag variables

Inertia animation variables

Listing 6.8 is the first part of our `ListPane` class, designed to house the `List` we created earlier. The exposed variables are quite straightforward:

- `content` is our `List`.
- `width` and `height` are the dimensions of the node. `width` is passed on to the `List`, where it's used to size the list items.
- `scrollY` is the scroll position of the `List` within our pane. The value is the `List` position relative to the `ListPane`, which is why it's negative. To scroll to pixel position 40, for example, we position the `List` at -40 compared to its container pane.

The private variables control the drag and the animation effect:

- To move the `List` we need to know how far we've dragged the mouse during this operation and where the `List` was before we started to drag. The private variable `clickY` records where inside the pane the mouse was when its button was pressed, and `scrollOrigin` records its scroll position at that time. `buttonDown` is a handy flag, recording whether or not we're in the middle of a drag operation.
- To create the inertia effect we must know how fast the mouse was traveling before its button was released, and `dragDelta` records that for us. We also need a `Timeline` for the effect, hence `dragTimeline`.
- If the `List` is smaller than the `ListPane`, we want to disable any scrolling or animation. The flag `noScroll` is used for this very purpose.

So much for the class variables. What about the actual scene graph and mouse event handlers? For those we need to look at listing 6.9.

Listing 6.9 ListPane.fx (part 2)

```
// ** Part 1 in listing 6.8
override function create() : Node {
    Group {
        content: [
            this.content ,
            Rectangle {
                width: bind this.width;
                height: bind this.height;
                opacity: 0.0;
                onMousePressed: function(ev:MouseEvent) {
                    animStop();
                    clickY = ev.y;
                    scrollOrigin = scrollY;
                    buttonDown = true;
                };
                onMouseDragged: function(ev:MouseEvent) {
                    def prevY = scrollY;
                    updateY(ev.y);
                    dragDelta = scrollY-prevY;
                };
                onMouseReleased: function(ev:MouseEvent) {
                    updateY(ev.y);
                    animStart(dragDelta);
                    dragDelta = 0;
```

List node

Background and mouse events

```
                    buttonDown = false;
                };
                onMouseWheelMoved: function(ev:MouseEvent) {
                    if(buttonDown == false) {
                        scrollY = restrainY (
                            scrollY + ev.wheelRotation
                                * content.cellWidth
                        );
                    }
                };
            }
        ];
        clip: Rectangle {
            x:0;  y:0;
            width: bind this.width;
            height: bind this.height;
        }
    }
}

function updateY(y:Number) : Void {
    if(noScroll) { return; }
    scrollY = restrainY( scrollOrigin-(y-clickY) );
}
function restrainY(y:Number) : Number {
    def h = content.layoutBounds.height-height;
    return
        if(y<0)  0
        else if(y>h)  h
        else  y;
}

function animStart(delta:Number) : Void {
    if(dragDelta>5 and dragDelta<-5) { return; }
    if(noScroll) { return; }

    def endY = restrainY(scrollY+delta*15);
    dragTimeline = Timeline {
        keyFrames: [
            KeyFrame {
                time: 1s;
                values: [
                    scrollY => endY
                        tween Interpolator.EASEOUT
                ];
            }
        ]
    };
    dragTimeline.playFromStart();
}
function animStop() : Void {
    if(dragTimeline!=null) {
         dragTimeline.stop();
    }
}
}
```

Constrain visible area

Limit scroll to list size

Inertia time line

The scene graph for `ListPane` consists of two nodes: the `List` itself and a `Rectangle` that handles our mouse events.

- When `onMousePressed` is triggered, we stop any inertia animation that may be running, store the initial mouse y coordinate and the current list scroll position, then flag the beginning of a drag operation.
- When `onMouseDragged` is called, we update the `List` scroll position and store the number of pixels we moved this update (used to calculate the speed of the inertia when we let go). The `restrainY()` function prevents the `List` from being scrolled off its top or bottom.
- When the `onMouseReleased` function is called, it updates the `List` position, kicks off the inertia animation, and resets the `dragDelta` and `buttonDown` variables so they're ready for next time.
- There's also a handler for the mouse scroll wheel, `onMouseWheelMoved()`, which should work only when we're not in the middle of a drag operation (we can drag or wheel, but not both at the same time!)

You'll note that the `Group` employs a `Rectangle` as a clipping area. Clipping areas are the way to show only a restricted view of a scene graph. Without this, the `List` nodes would spill outside the boundary of our `ListPane`. The clipping assignment creates the view port behavior our node requires, as demonstrated in figure 6.8.

Let's look at the `animStart()` function, which kicks off the inertia animation. The `delta` parameter is the number of pixels the pointer moved in the mouse-dragged event immediately before the button release. We use this to calculate how far the list will continue to travel. If the mouse movement was too slow (less than 5 pixels), or the `List` too small to scroll, we exit. Otherwise a `Timeline` animation is set up and started.

The list was our most ambitious piece of scene graph code yet. The result, complete with hover effect as the mouse moves over the list, is shown in figure 6.9. Even though it supports a lavish smooth scroll and animated reactions to the mouse pointer, it didn't take much more than a couple of hundred lines of code to write. It just shows how easy it is to create impressive UI code in JavaFX.

In the next section we'll delve into the exciting world of multimedia, as we plug our new list into the project application and use it to trigger video playback.

Figure 6.9 A closer look at our `List` and `ListPane`, with hover effect visible on the background of the list items

6.2.3 Using media in JavaFX

The time has come to learn how JavaFX handles media, such as the video files we'll be playing in our application. Before we look at the JavaFX Script code itself, let's invest time in learning about the theory. We'll start with figure 6.10.

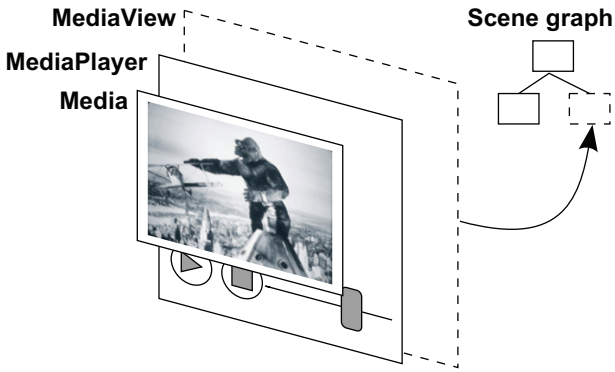

Figure 6.10 Like other JavaFX user interface elements, video is played via a dedicated `MediaView` scene graph node. (Note: `MediaPlayer` is not a visual element; the control icons are symbolic.)

To plug a video into the JavaFX scene graph takes three classes, located in the `javafx.scene.media` package. They are demonstrated in figure 6.10; starting from the outside, and working in, they are:

- The `MediaView` class, which acts as a bridge between the scene graph and any visual media that needs to be displayed within it. `MediaView` isn't needed to play audio-only media, because sound isn't displayed in the scene graph.
- The `MediaPlayer` class, which controls how the media is played; for example, stopping, restarting, skipping forward or backward, slowed down or sped up. `MediaPlayer` can be used to control audio or video. Important: `MediaPlayer` merely permits programmatic control of media; it provides no actual UI controls (figure 6.10 is symbolic). If you want play/pause/stop buttons, you must provide them yourself (and have them manipulate the `MediaPlayer` object).
- The `Media` class, which encapsulates the actual video and/or audio data to be played by the `MediaPlayer`.

As with images, JavaFX prefers to work with URLs rather than directly with local directory paths and filenames. If you read the API documentation, you'll see that the classes are designed to work with different types of media and to make allowances for data being streamed across a network.

The data formats supported fall into two categories. First, JavaFX will make use of the runtime operating system's media support, allowing it to play formats supported on the current platform. Second, for cross-platform applications JavaFX includes its own codec, available no matter what the capabilities of the underlying operating system.

Table 6.1 shows the support on different platforms. At the time this book was written, the details for Linux media support were not available, although the same mix of native and cross-platform codecs is expected.

The cross-platform video comes from a partnership deal Sun made with On2 for its Video VP6 decoder. On2 is best known for providing the software supporting Flash's own video decoder. The VP6 decoder plays FXM media on all JavaFX platforms, including mobile (and presumably TV too, when it arrives) without any extra

Table 6.1 JavaFX media support on various operating systems

Platform	Codecs	Formats
Mac OS X 10.4 and above (Core Video)	**Video:** H.261, H.263, and H.264 codecs. MPEG-1, MPEG-2, and MPEG-4 Video file formats and associated codecs (such as AVC). Sorenson Video 2 and 3 codecs. **Audio:** AIFF, MP3, WAV, MPEG-4 AAC Audio (.m4a, .m4b, .m4p), MIDI.	3GPP / 3GPP2, AVI, MOV, MP4, MP3
Windows XP/Vista (DirectShow)	**Video:** Windows Media Video, H264 (as an update). **Audio:** MPEG-1, MP3, Windows Media Audio, MIDI.	MP3, WAV, WMV, AVI, ASF
JavaFX (cross platform)	**Video:** On2 VP6. **Audio:** MP3.	FLV, FXM (Sun defined FLV subset), MP3

software installation. Regrettably, the only encoder for the On2 format at the time of writing seems to be On2 Flix, a proprietary commercial product.

Now that you understand the theory, let's push on to the final part of the project, where we build a working video player.

6.2.4 The Player class, version 2: video and linear gradients

We now have all the pieces; all that remains is to pull them together. The listing that follows is our largest single source file yet, almost 200 lines (be thankful this isn't a Java book, or it could have been 10 times that). I've broken it up into three parts, each dealing with different stages of the application. The opening part is listing 6.10.

Listing 6.10 Player.fx (version 2, part 1)

```
package jfxia.chapter6;

import javafx.geometry.HPos;
import javafx.geometry.VPos;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.control.Slider;
import javafx.scene.effect.Reflection;
import javafx.scene.input.MouseEvent;
import javafx.scene.layout.LayoutInfo;
import javafx.scene.layout.Stack;
import javafx.scene.media.Media;
import javafx.scene.media.MediaPlayer;
import javafx.scene.media.MediaView;
import javafx.scene.paint.Color;
import javafx.scene.paint.LinearGradient;
import javafx.scene.paint.Stop;
import javafx.scene.shape.Rectangle;
import javafx.scene.text.Font;
import javafx.scene.text.Text;
import javafx.scene.text.TextOrigin;
```

```
import javafx.stage.Stage;

import java.io.File;
import javax.swing.JFileChooser;

var sourceDir:File;
var sourceFiles:String[];

def fileDialog = new JFileChooser();
fileDialog.setFileSelectionMode(
   JFileChooser.DIRECTORIES_ONLY);
def ret = fileDialog.showOpenDialog(null);
if(ret == JFileChooser.APPROVE_OPTION) {
    sourceDir = fileDialog.getSelectedFile();
    if(sourceDir.isDirectory() == false) {
        println("{sourceDir} is not a directory");
        FX.exit();
    }
    def files:File[] = sourceDir.listFiles();
    for(i in [0 ..< sizeof files]) {
        def fn:String = files[i].getName().toLowerCase();
        if(fn.endsWith(".mpg") or fn.endsWith(".mpeg")
            or fn.endsWith(".wmv") or fn.endsWith(".flv")) {
            insert files[i].getName() into sourceFiles;
        }
    }
}
else {
    FX.exit();
}
// ** Part 2 is in listing 6.11; part 3 in listing 6.12
```

Select a directory

Check valid selection

Create video file list

When run, the program asks for a directory containing video files using Swing's own `JFileChooser` class. This time we're not using JavaFX wrappers around a Swing component; we're creating and using the raw Java class itself. Having created the chooser, we tell it to list only directories, then show it, and wait for it to return. Assuming the user selected a directory, we run through all its files, looking for potential videos based on their filename extension, populating the `sourceFiles` sequence when found.

Assuming we continue running past this piece of code, the next step (listing 6.11) is to set up the scene graph for our video player.

Listing 6.11 Player.fx (version 2, part 2)

```
// ** Part 1 is in listing 6.10; part 3 in listing 6.12
def margin = 10.0;
def videoWidth = 480.0;
def videoHeight = 320.0;
def reflectSize = 0.25;
def font = Font { name: "Helvetica";  size: 16; };
def listWidth = 200;
def listHeight =
    videoHeight*(1.0+reflectSize) +  margin*2;

var volumeSlider:Slider;
var balanceSlider:Slider;
```

Video display dimensions

Height matches media area

Volume/balance sliders

```
def list:ListPane = ListPane {
    content: List {
        content: sourceFiles;
        font: font;
        action: function(i:Integer) {
            player.media = Media {
                source: getVideoPath(i);
            }
            player.play();
        };
    };
    width: listWidth;
    height: listHeight;
}

var player:MediaPlayer = MediaPlayer {
    volume:  bind volumeSlider.value / 100.0;
    balance: bind balanceSlider.value / 100.0;
    onEndOfMedia: function() {
        player.currentTime = 0s;
    }
}
def view:Stack = Stack {
    layoutX: listWidth + margin;
    layoutY: margin;
    nodeHPos: HPos.CENTER;
    nodeVPos: VPos.BASELINE;
    content: [
        Rectangle {
            width: videoWidth;
            height: videoHeight;
            opacity: 0;
        } ,
        MediaView {
            fitWidth: videoWidth;
            fitHeight: videoHeight;
            preserveRatio: true;
            effect: Reflection {
                fraction: reflectSize;
                topOpacity: 0.25;
                bottomOpacity: 0.0;
            };
            mediaPlayer: player;
        }
    ]
}
def vidPos = bind player.currentTime.toSeconds() as Integer;

def panel:Group = Group {
    layoutY: listHeight;
    content: [
        Button {
            iconFilename: "play.png";
            action: function(ev:MouseEvent) {
                player.play();
            }
```

List display

Action: create then play media

Control video with player

Always rests on area baseline

Spacer rectangle

Reflection under video

Video position/duration (handy)

Control panel

Play button

```
        } ,
        Button {                                          <-- Pause button
            layoutX: 80;
            iconFilename: "pause.png";
            action: function(ev:MouseEvent) {
                player.pause();
            };
        } ,
        GridBox {                                         <-- GridBox layout
            layoutX:185;  layoutY:20;
            columns:4;
            nodeVPos: VPos.CENTER;
            horizontalGap:10;  verticalGap:20;
            content: [
                makeLabel("Volume") ,
                volumeSlider = Slider {                   <-- Volume slider
                    layoutInfo: LayoutInfo { width:150 }
                    value: 100;  max: 100;
                } ,

                makeLabel("Balance") ,
                balanceSlider = Slider {                  <-- Balance slider
                    layoutInfo: LayoutInfo { width:150 }
                    value: 0;  min: -100;  max: 100;
                } ,

                makeLabel("Playback") ,
                Text {                                    <-- Position display
                    content: bind
                        if(player.media!=null)
                            "{vidPos} seconds"
                        else
                            "No video";
                    font: font;
                    fill: Color.LIGHTBLUE;
                    textOrigin: TextOrigin.TOP;
                }
            ];
        }
    ];
};
// ** Part 1 is in listing 6.10; part 3 is in listing 6.12
```

In listing 6.11 we create the parts of our scene graph, which will be plugged into the application's `Stage` in part 3 (listing 6.12). There are three main parts: `list` is the scrolling list of videos from which the user can select, `view` is the video display area itself, and `panel` is the control panel that runs along the bottom of the window. You'll notice that the `MediaPlayer` is also given its own variable, `player`.

The list is constructed from the two classes, `List` and `ListPane`, we developed earlier. Its contents are the filenames from the directory returned by `JFileChooser`. The `action` event takes the selected list index and turns it into a URL (thanks to the `getVideoPath()` function we'll see later), creates a new `Media` object from it, plugs the `Media` object into `player`, and starts it playing.

The `player` node itself is quite simple. Its `volume` and `balance` variables are bound to sliders in the control panel, and it has an event handler (`onEndOfMedia`) that rewinds the video back to the start once it reaches the end.

The video area, `view`, uses another of JavaFX's standard layout nodes: `Stack`. `Stack` overlaps its children on top of each other, earlier nodes in the `content` sequence appearing below later nodes. Because children may be different sizes, the `nodeHPos` and `nodeVPos` properties determine how smaller nodes should be aligned. In our case we use a transparent spacer `Rectangle` to enforce the maximum size of our video area and then add the `MediaView` so it always rests against the bottom (baseline) of this area.

Figure 6.11 shows relationship of the key variables.

The script-level variables `videoWidth` and `videoHeight` determine the pixel size of the actual video node, `reflectSize` is the proportion of the node that gets reflected below it (see figures 6.1 and 6.7), and `margin` is the border around the whole area. Because the scrolling list extends for the full height of the video area, `listHeight` is calculated using `videoHeight`, including its `reflectSize`, plus the top and bottom `margin`.

The `fitWidth` and `fitHeight` parameters are set on the `MediaView` node, causing any video to be scaled to the `videoWidth`/`videoHeight` area, but `preserveRatio` is set so the video is never stretched out of proportion. Because a given video may be smaller, when scaled, than either `videoWidth` or `videoHeight`, we use the `Stack` node's `nodeHPos` and `nodeVPos` variables to fix the video centrally along the baseline of the area.

The reflection effect may look impressive, but in JavaFX applying any visual effect to a section of the scene graph is as easy as assigning the given node's `effect` variable. If you look in the API documentation for the `javafx.scene.effect` package, you'll see all manner of different effects you can apply; we'll be looking at more of them in future chapters. The `Reflection` effect adds a mirror below its node, with a given top/bottom opacity.

Finally, the control panel, aka the `panel` variable, will be familiar from version 1 of the project. The only substantial difference is that now the sliders are plugged into actual video player variables. The `makeLabel()` function is simply a convenience for creating label text; it appears in part 3 of the code. And speaking of part 3, here's listing 6.12.

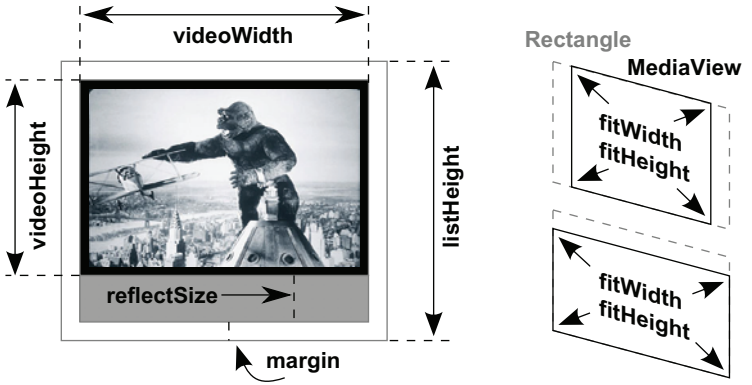

Figure 6.11 The video area layout and sizing are controlled by variables, some at the script level and others local to the scene graph node itself.

Listing 6.12 Player.fx (version 2, part 3)

```
// ** Part 1 is in listing 6.10; part 2 in listing 6.11
Stage {
    scene: Scene {
        content: [
            list,view,panel
        ];
        fill: LinearGradient {
            endX:0;  endY:1;
            proportional: true;
            stops: [
                Stop {
                    offset:0.55;  color:Color.BLACK;
                } ,
                Stop {
                    offset:1;  color:Color.STEELBLUE;
                }
            ];
        };
    };
    title: "Player v2";
    resizable: false;
}

function makeLabel(str:String) : Text {
    Text {
        content: str;
        font: font;
        fill: Color.LIGHTBLUE;
        textOrigin: TextOrigin.TOP;
    };
}

function getVideoPath(i:Integer) : String {
    def f = new File(sourceDir,sourceFiles[i]);
    return f.toURI().toString();
}
```

Scene graph bits

Linear gradient background

Handy label maker function

Convert list filename to URL

The final part of our application. Whew!

We see here that the three nodes we created in the second part (`list`, `view`, and `panel`) are hooked into our scene. At the bottom of the listing we see the `makeLabel()` and `getVideoPath()` convenience functions we used previously. But what's that in the middle of the listing, plugged into the scene's `fill` parameter? That's a `LinearGradient`, and it's responsible for the graduated fill color that sits behind the whole window's contents. If you think it looks rather odd, don't worry; I've devoted an entire section to unlocking its secrets, up next.

6.2.5 Creating varying color fills with LinearGradient

Instead of using a boring flat color for the window background, in listing 6.12 we create a `LinearGradient` and use it as the scene's fill. We can do this because the `Scene` class accepts a `javafx.scene.paint.Paint` instance as its background fill. The `Paint` class is

a base for any object that can be used to determine how the pixels in a shape will be drawn; the flat colors we used in previous examples are also types of `Paint`, albeit not very exciting ones.

A gradient paint is one that transitions between a set of colors as it draws a shape. Good examples of linear gradients might include a color spectrum or a chrome metal effect, as shown in figure 6.12.

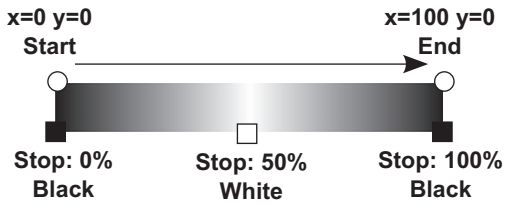

Figure 6.12 A gradient paint is one where the pixel tone changes over the course of a given area. Colors are set at stops along a line, and the paint transitions between them as a shape is drawn.

Think about how the color gradually changes across the painted area. To define a linear gradient we need two things: a line with start and end points (a path to follow) and a list of colors on the line plus where they are positioned. For a simple rainbow spectrum we might define a horizontal (or vertical) line, with each color stop spaced equally along its length.

The line can function either with proportional sizing or via absolute pixel coordinates. What's the difference? Take a look at the examples in figure 6.13.

When `proportional` is true, the line coordinates (`startX`, `startY`, `endX`, and `endY`) are scaled across a virtual coordinate space from 0 to 1, which is stretched to fit the actual painted area whenever the paint is applied. Without `proportional` set, the line start and end coordinates are absolute pixel positions. This means if you define a vertical gradient line of (0,0) to (0,100) but then paint an area of 200 x 200 pixels, the transition will not cover the entire painted area. By setting a parameter called `cycleMethod` we can control how the remainder will paint. The default option extends the color at either end of the gradient line. Alternatively we can repeat the gradient or reflect it by painting it backwards.

Incidentally, `LinearGradient` creates stripes of color along a gradient line, but you might also want to check out its cousin, `RadialGradient`, which paints circular patterns. It can be particularly useful for creating pleasing 3D ball effects.

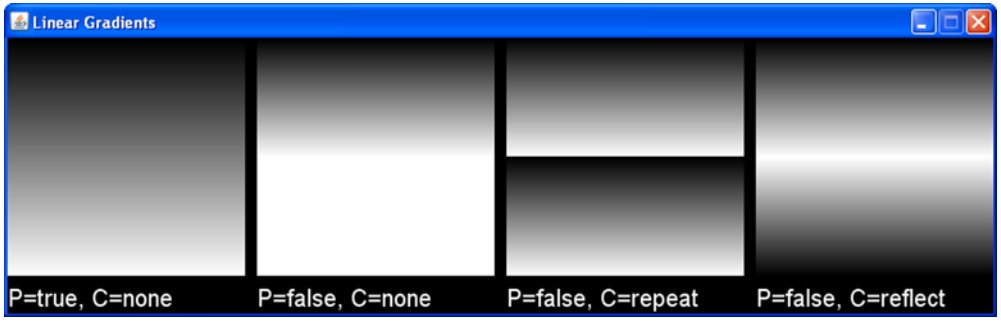

Figure 6.13 A proportional (P) gradient scaled to full height. Then three nonproportional examples, gradient (0,0) to (0,100), painted onto 200 x 200 sized rectangles with various cycle (C) methods.

6.2.6 Running version 2

Running version 2 of the project gives us our video player. Selecting a file from the list on the left will play it in the central area, complete with snazzy reflection over the shaded floor. Figure 6.14 shows what you should expect when firing up the application, selecting a directory, and playing a video (especially if you're a Mighty Reds fan).

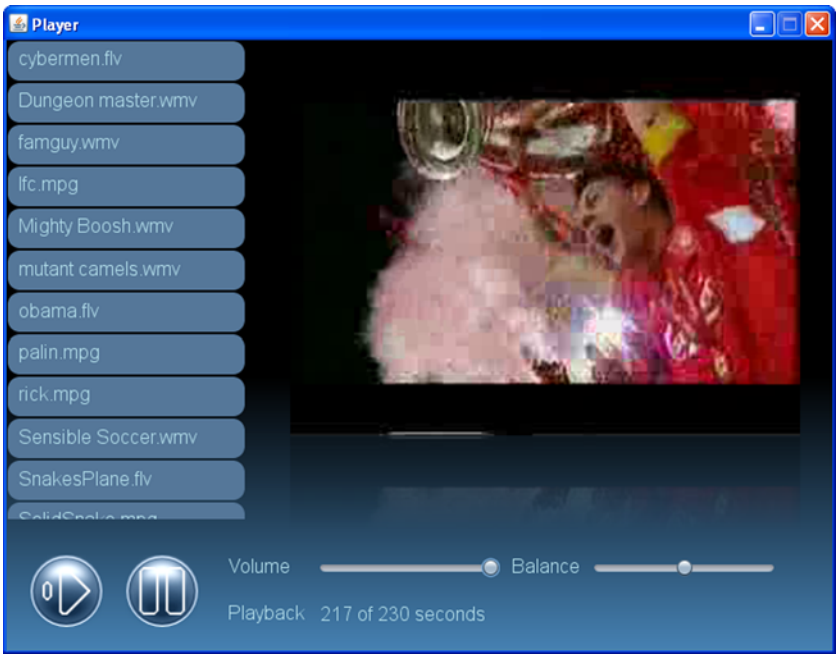

Figure 6.14 You'll never walk alone: relive favorite moments with your own homemade video player (like your soccer team lifting the Champion's League trophy).

Okay, so perhaps it's not the most functional player in the world, but it's still very impressive when you consider how little work we needed to pull it off. Think how long it would have taken to code all the UI and effects using Java or C++. And that's before we even *think* about getting video playback working.

6.3 Bonus: taking control of fonts

Time for another detour section. This one doesn't directly relate to the material in this chapter, but it's a very useful UI trick, and this seems an opportune moment to mention it.

Like Java, JavaFX has a small set of standard fonts with dependable names and appearances. It can also use fonts available on the local computer, although there's no guarantee which fonts will be installed or precisely what they'll look like. So, what if we need a specific font, not guaranteed to be available on every computer? Fortunately,

JavaFX allows us to embed fonts directly inside our application, guaranteeing availability and appearance. Figure 6.15 shows an example.

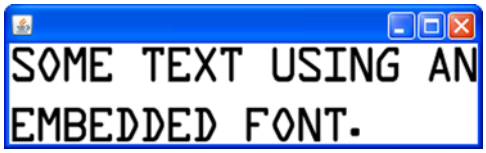

Figure 6.15 Text rendered using an embedded TrueType font file

To embed a typeface we need a font file in a format such as TrueType. The file is not loaded directly from a URL or an input stream but assigned an alias from which it can be referenced like any standard or operating system font. This is a two-step process:

1. The font file should be placed somewhere under the build directory of your project, so it is effectively *inside* the application (and ultimately inside the application's JAR file, when we package it—see chapter 9).
2. A file named META-INF/fonts.mf should be created/edited. This will provide mappings between font names and their associated embedded files. The META-INF directory should be familiar to you as the place where Java stores metadata about resources and JARs. Most JAR files have a META-INF directory, often created by the `jar` utility if not already part of the project.

When a JavaFX application needs to resolve a font name, it first checks the embedded fonts for a match; then it checks the standard JavaFX fonts and then the operating system's fonts. If no match can be found, it uses a default fallback font. So, by creating a fonts.mf file we can get our own fonts used in preference to other fonts, but what does this file look like? Listing 6.13 demonstrates.

Listing 6.13 fonts.mf

```
Data\ Control = /ttf/data-latin.ttf
```

This single line creates a mapping between the name `Data Control` (the backslash escapes the space) and a TrueType font file called data-latin.ttf, living in a directory called ttf off the build directory. Having created a font mapping, we can reference it in our code like any other font, as shown in listing 6.14.

Listing 6.14 FontTest.fx

```
import javafx.scene.Scene;
import javafx.scene.text.*;
import javafx.stage.Stage;

Stage {
    scene: Scene {
        content: Text {
            textOrigin: TextOrigin.TOP;
            font: Font {
                name: "Data Control";
                size: 40;
            }
            content: "Some text using an\nembedded font.";
```

```
            }
        }
}
```

Listing 6.14 is a simple program consisting of a text node in a window. It looks like figure 6.15 when run. The `Font` declaration references `Data Control` like it was a standard JavaFX or operating system font, which resolves via the fonts.mf file to our embedded font. When compiled, our application's build directory should feature the following files:

- FontTest.class
- META-INF/fonts.mf
- ttf/data-latin.ttf

The FontTest.fx source didn't specify a package, so its class file will compile into the root (I've omitted compiler implementation classes for readability). Also off the root is the META-INF directory with our font mapping file and the ttf directory where we deposited the font data file. Actually, it doesn't matter where we put the font file, so long as it lives under the build directory and the correct location is noted in the mapping file.

With this simple technique we can bundle any unusual fonts we need inside the application (and ultimately inside its JAR file), guaranteeing that they are available no matter where our code runs. A very useful UI trick indeed!

Credit where credit's due

I'm indebted to Rakesh Menon, who was the first (as far as I know) to reveal this method of embedding fonts into a JavaFX application. His blog post is located here:

http://blogs.sun.com/rakeshmenonp/entry/javafx_custom_fonts

6.4 Summary

In this chapter we've seen in greater depth how to use the standard scene graph nodes to build fairly sophisticated pieces of UI, we've looked at how to include images and video in our JavaFX applications, and we've played around with gradient paints. We've also seen our first example of plugging an effect (reflection) into the scene graph.

Writing good scene graph code is all about planning. JavaFX gives you some powerful raw building blocks; you have to consider the best way to fit them together for the effect you're trying to achieve. Always be aware that a scene graph is a different beast than something like Swing or Java 2D. It's a structured representation of the graphics on screen, and as such we need to ensure it includes layout and spacing information, because we don't have direct control of the actual pixel painting as we do in Java 2D. Transparent shapes can be used to enforce spacing around parts of our scene graph, but they can also be used as a central event target.

Hopefully the source code in this chapter has given you ideas about how to write your own UI nodes. Feel free to experiment with the player, filling in its gaps and adding new features. Or take the custom nodes and develop them further in your own applications.

In the next chapter we're shifting focus from simple interactive scene graph nodes to full blown controls, as we explore JavaFX's standard user interface APIs. We'll also discover another powerful way to customize node layout.

But for now, enjoy the movie!