10 Clever graphics and smart phones

The chapter covers

- Constructing complex scene graphs
- Handling device-agnostic key input
- Going mobile, with the phone emulator
- Tuning apps, for better performance

Previous projects in this book have introduced a lot of fresh material and concepts, covering the core of the JavaFX Script language and its associated JavaFX APIs. Although we haven't explored every nook and cranny of JavaFX, by now you should have experienced a representative enough sample to navigate the remainder of the API without getting lost or confused. So this chapter is a deliberate shift down a gear or two.

Over the coming pages we won't be discovering any new libraries, classes, or techniques. Instead, this chapter's project will focus on a couple of goals. We'll start by reinforcing the skills you've acquired thus far, pushing the scene graph in directions you've previously not seen, and then end by moving our finished project onto the JavaFX Mobile platform.

This chapter is being written against JavaFX release 1.2 (June 2009), and, regrettably, at the time of this writing, the mobile emulator is bundled only with the Windows JavaFX SDK. This is an omission the JFX team is committed to address soon; hopefully the fruits of their labor will be available by the time this book reaches you. Despite the limitations of the mobile preview, there's still a lot of value in the project. The code has been written to run in both the desktop and the mobile environments, so non-Microsoft users can work through the majority of the project while waiting for the emulator to arrive on their platform.

The scene graph code we'll soon encounter is by far our most complex and imaginative yet, so once I've rattled through an outline of the project, we'll spend a little time on the secrets of how it's put together. But, first of all, we need to know what the project is.

10.1 Amazing games: a retro 3D puzzle

It's incredible to think, when looking at the sophistication of consoles like the PlayStation and Xbox, just how far graphics have come in the last few decades. I'm just about old enough to remember the thrill of the Atari 2600 and the novelty of controlling an image on the TV set. The games back then seemed every bit as exciting as the games today, although having replayed some of my childhood favorites thanks to modern-day emulators, I'm at a loss to explain why. Some games never lose their charm, however. I discovered this the hard way, after losing the best part of a day to a Rubik's Cube I discovered while clearing out old junk.

In this chapter we're going to develop a classic 3D maze game, like the ones that sprang up on the 8-bit computers a quarter of a decade before JavaFX was even a twinkle in Chris Oliver's eye. Retro games are fun, and they're popular with phone owners; not only is there an undeniable nostalgia factor, but their clean and simple game play fits nicely with the short bursts of activity typical of mobile device usage.

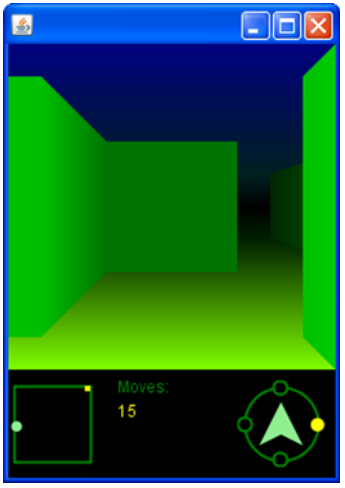

Figure 10.1 Get lost! This is our simple 3D maze game. The whole thing is constructed from the JavaFX scene graph, using basic shapes.

Figure 10.1 shows our retro maze game in action. The maze is really a grid, with some of the cells as walls and others as passageways. Moving through the maze is done one cell at a time, either forward, backward, left, or right. It's also possible to turn 90 degrees clockwise or counterclockwise. There's no swish true-3D movement here; the display simply jumps one block at a time, in true Dungeon Master fashion.

To aid the player we'll add a radar, similarly retro fashioned, giving an idea of where the player is in relation to the boundaries of the maze. We'll also add a compass, spinning to show which way the

view is facing. A game needs a score, so we'll track the number of moves the player has made, the objective being to solve the maze in as few moves as possible.

The interface looks fairly simple, and that's because I want to spend most of the pages ahead concentrating on the 3D part of the game rather than on mundane control panel components. The 3D effect is quite an unusual use of the scene graph, and it demands careful forward thinking and node management. Let's start by looking at the theory behind how it works.

10.1.1 Creating a faux 3D effect

One evening, many years ago, I wandered up the road where I lived at the time, and found myself transported back to Victorian London. A most surreal moment! It happened that a film crew had spent the morning shooting scenes for an episode of *Sherlock Holmes*, and the nearby terrace housing had undergone a period makeover. If you've ever visited a Hollywood-style back lot, you'll be familiar with how looks can deceive. Everything from the brick walls to the paved sidewalks is nothing more than lightweight fabrications, painted and distressed to make them look authentic.

This may surprise some readers, but the walls in our project do not use any mind-bending 3D geometry. The maze is nothing more than the illusion of 3D, creating the effect without any of the heavy lifting or number crunching. Now it's time to reveal its secrets.

Picture five squares laid out side by side in a row. Now picture another row of squares, this time twice the size, overlaid and centered atop our original squares. And a third row, again twice as big (making them four times the size of the original row), and a fourth row (eight times the original size), all overlaid and centered onto the scene. If we joined the points from all these boxes, we might get a geometry like the one in figure 10.2.

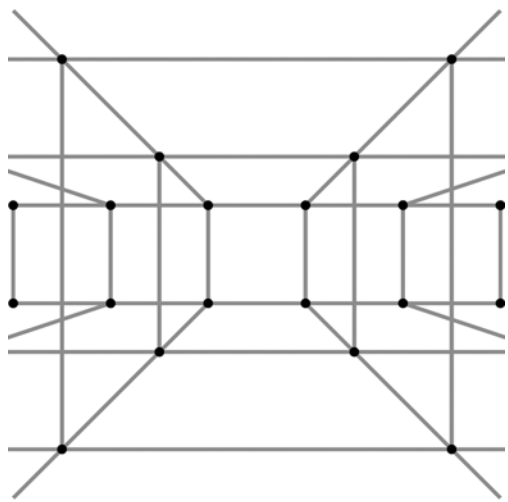

Figure 10.2 The 3D in our maze is all fake. The grid defines the maze geometry without using any complex mathematics.

Figure 10.2 has been clipped so that some of the squares are incomplete or missing. We can see the smallest row of squares, five in all, across the middle of the figure. The next row has three larger squares, the extremes of which are partially clipped. There are three even larger squares, so big only one fits fully within the clipped area, with just a fragment of its companions showing. And the final row of squares is so large they all fall entirely outside the clipping area, but we can see evidence for them in the diagonal

lines leading off at the far corners of the figure. This collection of squares is all we need to construct the illusion of three dimensions.

10.1.2 Using 2D to create 3D

So, the 3D effect in our maze is entirely constructed from a plain and simple 2D grid, but how does that actually work? Figure 10.3 shows how the geometry of our 3D display is mapped to the 20 x 20 grid we're using to represent it.

By using rows of squares and connecting their points, we can build a faux 3D view. The visible area exposed in figure 10.2 fits within a grid, 20 x 20 points. Using a grid of that size we can express the coordinates of every square using integer values only. Figure 10.3 shows how those points are distributed inside (and outside) the 20 x 20 grid. Remember: even though the diagram *looks* 3D, the beauty is we're still dealing with 2D x/y points.

- Our five smallest squares (let's say they're the *farthest away*) are each 4 points wide, with the first one having its top-left and bottom-right coordinates at (0,8)-(4,12), the second at (4,8)-(8,12), and so on.
- The next row of three squares is 8 points in size: (-2,6)-(6,14), then (6,6)-(14,14), and finally (14,6)-(22,14). The first and the last squares fall slightly outside the (0,0)-(20,20) viewport.
- The next row of three is 16 points in size: (-14,2)-(2,18), then (2,2)-(18,18), etc. The first and last squares fall predominantly outside the clipping viewport.
- The final row, not shown in figure 10.3, needs only one square, which falls entirely outside the clipped area, at (-6,-6)-(26,26).

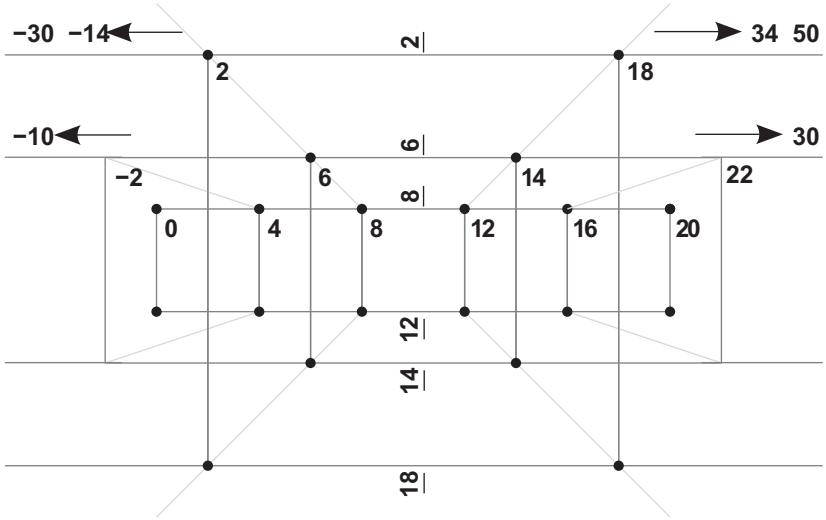

Figure 10.3 The geometry of our maze. Using a flat 20 x 20 grid as the viewport, the regular numbers describe x coordinates, and the rotated numbers (underlined) describe y coordinates.

This simple grid of points is the key behind the custom node that creates the 3D maze display. Now that we know the theory, we need to see how it works using code.

10.2 The maze game

The game we'll develop will be constructed from custom nodes, centered on a *model* class for the maze and player status. The scene graph code is simple, except for the 3D node, which is highly complex. The 3D custom node demonstrates just how far we can push JavaFX's *retained mode* graphics, with clever positioning and management of nodes, to create a display more typical of *immediate mode* (the model Swing and Java 2D use).

10.2.1 The MazeDisplay class: 3D view from 2D points

We might as well start with the most complex part of the UI, indeed probably the most involved piece of UI code you'll see in this book. The `MazeDisplay` class is the primary custom node responsible for rendering the 3D view. As with previous source code listings, this one has been broken into stages. The first of these is listing 10.1.

Listing 10.1 MazeDisplay.fx (part 1)

```
package jfxia.chapter10;

import javafx.scene.CustomNode;
import javafx.scene.Group;
import javafx.scene.Node;
import javafx.scene.paint.Color;
import javafx.scene.paint.LinearGradient;
import javafx.scene.paint.Stop;
import javafx.scene.shape.Polygon;
import javafx.scene.shape.Rectangle;

package class MapDisplay extends CustomNode {
    def xPos:Integer[] = [
        0,  4,  8, 12, 16, 20 ,
      -10, -2,  6, 14, 22, 30 ,
      -30,-14,  2, 18, 34, 50 ,
      -70,-38, -6, 26, 58, 90
    ];
    def yPos:Integer[] = [
        8,  6,  2, -6 ,
       12, 14, 18, 26
    ];

    public-init var map:Map;

    var wallVisible:Boolean[];
    def scale:Integer = 12;
// Part 2 is listing 10.2
```

Horizontal points on faux 3D

Vertical points on faux 3D

Map data

Manipulates scene graph

Scale points on screen

We have the opening to our maze view class, and already we're seeing some pretty crucial code for making the 3D effect work.

The opening tables, `xPos` and `yPos`, represent the points on the 20 x 20 grid shown in figure 10.3. Each line in the `xPos` table represents the horizontal coordinate for five

squares. Even though *nearer* (larger) rows require only three squares or one square, we use five consistently to balance the table and make it easier to access. Each line contains six entries because we need not just the left-side coordinate but the right side too; the final entry defines the right-hand side of the final square. The first line in the table represents the row of squares farthest away (smallest), with each successive line describing nearer (larger) rows.

The `yPos` table documents the vertical coordinate for each row. The first line describes the y positions for the tops of the farthest (smallest) row, through to the nearest (largest) row. The next line does the same for the bottoms. We'll be using the `xPos` and `yPos` tables when we build our scene graph, shortly. Meanwhile, let's consider the remainder of the variables:

- `map` variable is where we get our maze data from. It's an instance of the `Map` class, which we'll see later.
- The `wallVisible` boolean sequence is used to manipulate the scene graph once it has been created.
- `scale` decides how many pixels each point in our 20 x 20 grid is worth. Setting `scale` to 12 results in a maze display of 240 x 240 pixels, ideal for our mobile environment. Larger or smaller values allow us to size the maze to other display dimensions.

The next step is to create the application's scene graph. Listing 10.2 shows how it fits together.

Listing 10.2 MazeDisplay.fx (part 2)

```
// Part 1 is listing 10.1
    override function create() : Node {
        def n:Node = Group {
            content: [
                Rectangle {
                    width: 20*scale;
                    height: 20*scale;
                    fill: LinearGradient {
                        proportional: true;
                        endX: 0.0;  endY: 1.0;
                        stops: [
                            Stop { offset: 0.0;
                                color: Color.color(0,0,.5); },
                            Stop { offset: 0.40;
                                color: Color.color(0,.125,.125); },
                            Stop { offset: 0.50;
                                color: Color.color(0,0,0); },
                            Stop { offset: 0.60;
                                color: Color.color(0,.125,0); },
                            Stop { offset: 1.0;
                                color: Color.color(.5,1,0); }
                        ];
                    };
                } ,
```

Sky and floor gradient

```
                _wallFront(0,4,0 , 0.15),
                _wallSides(1,4,0 , 0.15,0.45),
                _wallFront(1,3,1 , 0.45),
                _wallSides(2,3,1 , 0.45,0.75),
                _wallFront(1,3,2 , 0.75) ,
                _wallSides(2,3,2 , 0.75,1.00)
            ]
            clip: Rectangle {
                width: 20*scale;
                height: 20*scale;
            };
            cache: true;
        };
        update();

        n;
    }
// Part 3 is listing 10.3
```

Smallest row, fronts

Medium row, sides/fronts

Large row, sides/fronts

X-large sides

Cache output

Populate scene with map data

The `create()` function should, by now, be immediately recognizable as the code that creates our scene graph. It begins with a simple background `Rectangle`, using a `LinearGradient` to paint the sky and floor, gradually fading off to darkness as they approach the horizon. To populate the colors in the gradient we're using a script-level (static) function of `Color`, which returns a hue based on red, green, and blue values expressed as decimals between 0 and 1.

After the background, a series of function calls populates the graph with front-facing and side-facing walls. These are convenience functions that help keep `create()` looking nice and clean (I tend to prefix the name of such *refactored* functions with an underscore, although that's just personal style). We'll look at the mechanics of how the shapes are added to the scene in a moment, but for now let's just focus on the calls themselves. Here they are again:

```
_wallFront(0,4,0 , 0.15),
_wallSides(1,4,0 , 0.15,0.45),
_wallFront(1,3,1 , 0.45),
_wallSides(2,3,1 , 0.45,0.75),
_wallFront(1,3,2 , 0.75) ,
_wallSides(2,3,2 , 0.75,1.00)
```

The first call is to `_wallFront()`. It adds in the back row of five front-facing walls, which you can see in figure 10.4. The three-integer parameters all relate to the tables we saw in listing 10.1. The first two parameters are delimiters, from 0 to 4 (five walls), and the third parameter determines which parts of the `xPos` and `yPos` tables to use for coordinates. In this case we're using line 0 (the first six entries) in `xPos` and entry 0 in the top/bottom lines from `yPos`. In plain English this means we'll be drawing boxes 0 to 4 using horizontal coordinates 0, 4, 8, 12, 16, and 20 from `xPos` and vertical coordinates 8 and 12 from `yPos`. The final, fractional parameter determines how dark to make each wall. Because these walls are the farthest away, they are set to a very low brightness.

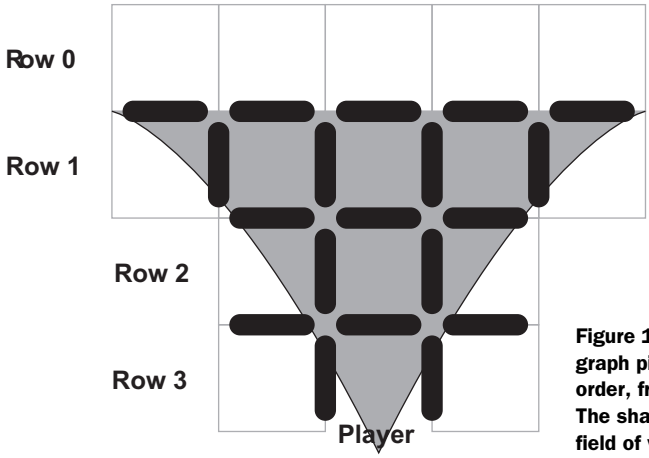

Figure 10.4 A plan view of the scene graph pieces that have to be added, in order, from back (row 0) to front (row 3). The shaded area represents the player's field of view.

The next line is a call to `_wallSides()`. It adds in the four side-facing walls, two left of center, two right of center. The first three parameters do pretty much the same thing, but there are two brightness parameters. This is because the *perspective* walls (the ones running *into* the screen) have a different brightness from their farthest to their nearest edge. The first parameter is for the farthest; the second is for the nearest.

The remaining function calls add in the other front and side walls, working from the back to the front of the 3D scene. Next, we see the actual function code (listing 10.3).

Listing 10.3 MazeDisplay.fx (part 3)

```
// Part 1 is listing 10.1; part 2 is listing 10.2
    function _wallFront(x1:Integer,x2:Integer , r:Integer,
    op:Number ) : Node[] {
        for(x in [x1..x2]) {                    <-- For each wall
            insert false into wallVisible;
            def pos:Integer = sizeof wallVisible -1;   Add on/off switch

            Polygon {                           <-- Front wall polygon
                points: [
                    xPos[r*6+x+0]*scale , yPos[0+r]*scale , // UL
                    xPos[r*6+x+1]*scale , yPos[0+r]*scale , // UR
                    xPos[r*6+x+1]*scale , yPos[4+r]*scale , // LR
                    xPos[r*6+x+0]*scale , yPos[4+r]*scale   // LL
                ];
                fill: Color.color(0,op,0);
                visible: bind wallVisible[pos];     <-- Bind to on/off switch
            };
        }
    }

    function _wallSides(x1:Integer,x2:Integer , r:Integer,
    opBack:Number,opFore:Number) : Node[] {
        var half:Integer = x1 + ((x2-x1)/2).intValue();   For each wall
        for(x in [x1..x2]) {
```

```
                def rL:Integer = if(x>half) r else r+1;
                def rR:Integer = if(x>half) r+1 else r;
                def opL:Number = if(x>half) opBack else opFore;
                def opR:Number = if(x>half) opFore else opBack;

                insert false into wallVisible;
                def pos:Integer = sizeof wallVisible -1;

                Polygon {
                    points: [
                        xPos[rL*6+x]*scale , yPos[0+rL]*scale , // UL
                        xPos[rR*6+x]*scale , yPos[0+rR]*scale , // UR
                        xPos[rR*6+x]*scale , yPos[4+rR]*scale , // LR
                        xPos[rL*6+x]*scale , yPos[4+rL]*scale   // LL
                    ];
                    fill: LinearGradient {
                        proportional: true;
                        endX:1;  endY:0;
                        stops: [
                            Stop {
                                offset:0;
                                color:Color.color(0,opL,0);
                            } ,
                            Stop {
                                offset:1;
                                color:Color.color(0,opR,0);
                            }
                        ];
                    };
                    visible: bind wallVisible[pos];
                };
            }
        }
// Part 4 is listing 10.4
```

Near/far edge?

Near/far brightness?

Add on/off switch

Side wall polygon

Left edge color

Right edge color

Bind to on/off switch

Listing 10.3 shows the two functions responsible for adding the walls into our scene graph. The first function, `_wallFront()`, adds the front-facing walls. It loops inclusively between two delimiters, parameters `x1` and `x2`, adding `Polygon` shapes. The `r` parameter determines which row in the view we're drawing, which in turn determines the parts of the `xPos` and `yPos` tables we should use. For example, when `r = 0` the table data for the farthest (smallest) row of walls is used; when `r = 3` the nearest (largest) row is used.

The polygon forms a square using the coordinate pairs upper-left, upper-right, lower-right, and lower-left, in that order. We could have used a `Rectangle`, but using a `Polygon` makes the code more consistent with its companion `_wallSides()` function.

Because the `xPos` and `yPos` tables are, in reality, linear sequences, we need to do a little bit of math to find the correct value. Each row of horizontal coordinates in `xPos` has six points, from left to right across the view. There are two sets of vertical coordinates (upper y and lower y) in `yPos`, each with four points (rows 0 to 3). For horizontal coordinates we multiply `r` up to the right row and then add on the wall index to find the left-hand side or its next-door neighbor for the right-hand side. The vertical coordinates, which define the top and bottom of the shape, are as simple as reading the `r`th value from the first and second line of `yPos`.

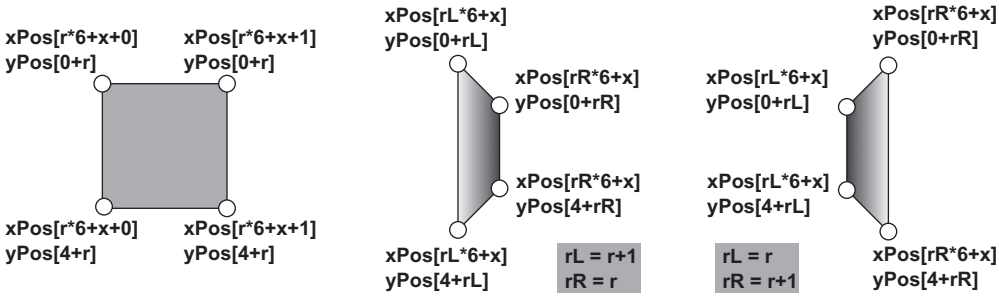

Figure 10.5 By plotting the points on our polygon, using the `xPos` and `yPos` tables for reference, we can created the illusion of perspective.

Figure 10.5 shows how the x and y coordinates are looked up inside the two tables. Remember, once we've used the tables to find the point in our 20 x 20 grid, we need to apply `scale` to multiply it up to screen-pixel coordinates.

Just before we create the polygon, we add a new boolean to `wallVisible`, and in the polygon itself we bind visibility to its index. This allows us to switch the wall on or off (visible or invisible) later in the code. This switch is key to updating the 3D view, as demonstrated in the final part of the code, in listing 10.4.

The `_wallSides()` function is a more complex variation of the function we've just studied. This time, one edge of the polygon is on one row, and the other edge is on another row. The left and right edges of the polygon are positioned depending on whether we're drawing a wall on the left side of the view or the right, in order to create the illusion of perspective. The `half` variable is used for that purpose, and instead of one `r` value we have two: one for the left and one for the right. We also have two color values for the edges of the polygon, to ensure it gets darker as it goes *deeper* into the display.

Once again we add a boolean to `wallVisible` and bind the polygon's visibility to it. But what do we do with all these booleans? The answer is in listing 10.4.

Listing 10.4 MazeDisplay.fx (part 4)

```
// Part 1 is listing 10.1; part 2, listing 10.2; part 3, listing 10.3
    package function update() : Void {
        def walls:Integer[] = [
            -2, 2,-3  ,
            -2,-1,-2  ,  1, 2,-2 ,
            -1, 1,-2  ,
            -1,-1,-1  ,  1, 1,-1 ,
            -1, 1,-1  ,
            -1,-1, 0  ,  1, 1, 0
        ];

        var idx:Integer = 0;
        var pos:Integer = 0;
        while(idx<sizeof walls) {
            var yOff:Integer = walls[idx+2];
            for(xOff in [walls[idx]..walls[idx+1]]) {
```

Row 0, fronts
Row 1, sides
Row 1, fronts
Row 2, sides
Row 2, fronts
Row 3, sides
(x1,y) to (x2,y)

```
                var rot:Integer[] = map.rotateToView(xOff,yOff);
                wallVisible[pos] = not map.isEmpty                    Change
                    (map.x+rot[0] , map.y+rot[1]);                    visibility
                pos++;
            }
            idx+=3;                    Next x1, x2,
        }                              y triple
    }
}
```

This is our final chunk of the mammoth 3D code. Having put all our polygons in place, we need to update the 3D view by switching their visibility on or off, depending on which cells of the map inside our viewport are walls and which are empty (passageways). We do this each time the player moves, using the `update()` function.

A reminder: we added rows of polygons into our scene graph, in sequence, from farthest to nearest. At the same time we added booleans to `wallVisible` in the same order, tying them to the corresponding polygon's visibility. These booleans are the means by which we will now manipulate each wall polygon. As we move around the maze, different parts of the map fall inside our field of view (the viewport). The wall nodes never get moved or deleted, and when a given cell of the map has no wall block present, the corresponding polygons are simply made invisible (they're still there in the graph; they just don't get drawn).

The `update()` function looks at map positions, relative to the player's position and orientation (facing north, south, east, or west), and sets the visibility boolean on the corresponding node of the scene graph. Figure 10.6 shows how the relative coordinates relate to the player's position at (0,0).

We need to work through each polygon node in the scene graph, relate it to an x/y coordinate in the map, and set the corresponding `wallVisible` boolean to turn the polygon on or off. The `walls` table at the start of the code holds sets of x1, x2, and y

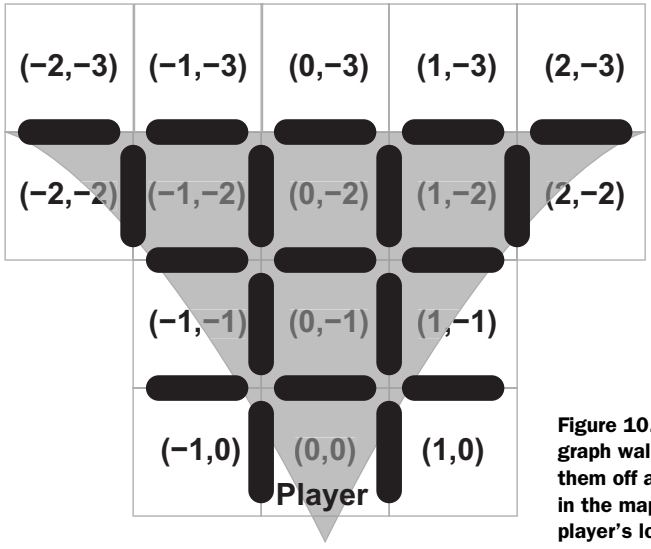

Figure 10.6 Having created our scene graph walls, we need to be able to switch them off and on depending on which cells in the map are wall blocks, relative to the player's location.

triples that control this process. Let's remind ourselves of the table, with one eye on figure 10.6, so we can see what that means:

```
def walls:Integer[] =
[   -2, 2,-3  ,
    -2,-1,-2  ,  1, 2,-2 ,
    // Snipped
]
```

The first nodes in the scene graph are the back walls, which run from coordinates (-2,-3) to (2,-3) relative to our player's position, where (0,0) is the cell the player is currently standing on. This is why our first triple is -2,2,-3 (x = -2 to x = 2, with y = -3).

Next we have four side walls, controlled by cells (-2,-2) and (-1,-2) on the left-hand side of the view and (1,-2) and (2,-2) on the right. You can see that the second line of `walls` has the corresponding triples. (We don't bother with the central cell, (0,-2), because that cell's side walls are entirely obscured by its front-facing polygon.)

The remaining lines in `walls` deal with the rest of the scene graph, row by row. All of these values are fed into a loop, translating the table into on/off settings for each wall polygon.

```
var idx:Integer = 0;
var pos:Integer = 0;
while(idx<sizeof walls) {
    var yOff:Integer = walls[idx+2];
    for(xOff in [walls[idx]..walls[idx+1]]) {
        var rot:Integer[] = map.rotateToView(xOff,yOff);
        wallVisible[pos] = not map.isEmpty
            (map.x+rot[0] , map.y+rot[1]);
        pos++;
    }
    idx+=3;
}
```

The `while` loop works its way through the `walls` table, three entries at a time; the variable `idx` stores the current offset into `walls`. Remember, the first value in the triple is the start x coordinate, the second is the end x coordinate, and the third is the y coordinate. The nested `for` loop then works through these coordinates, rotating them to match the player's direction and adding them into the current player x/y position to get the absolute cell in the map to query. We then query the cell to see if it is empty or not.

Once this code has run, each of the polygons in the scene graph will be visible or invisible, depending on its corresponding entry in the map. And voilà, our 3D display is alive!

I did warn you this was the most complex piece of scene graph code in the book, and I'm sure it didn't disappoint. You may have to scan the details a couple of times just to ensure they sink in. The purpose behind this example is to show that retained mode graphics (scene graph-based, in other words) don't have to be simple shapes. They are capable of complexity similar to immediate-mode graphics. All it takes is a

little planning and imagination. You'll be glad to hear the remaining custom nodes are trivial by comparison.

10.2.2 The Map class: where are we?

Now that we've seen the `MapDisplay` class, it's time to introduce the *model* class that provides its data. Listing 10.5 provides the code.

Listing 10.5 Map.fx (part 1)

```
package jfxia.chapter10;

package class Map
{   def wallMap:Integer[] = [
        1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1 ,
        1,0,0,1,0,1,0,0,0,0,0,0,1,0,1 ,
        1,1,0,1,0,1,1,1,0,1,1,0,1,1,0,1 ,
        1,0,0,1,0,1,0,1,0,0,0,0,0,1,0,1 ,
        1,0,1,1,0,1,0,1,0,1,1,1,0,0,0,1 ,
        1,0,0,1,0,0,0,1,0,1,0,1,1,1,1,1 ,
        1,0,0,1,1,1,1,1,0,1,0,0,0,0,0,1 ,
        1,0,1,1,0,0,0,1,0,1,1,1,0,1,0,1 ,
        1,0,0,1,1,0,1,1,0,0,0,1,0,1,1,1 ,
        1,1,0,0,0,0,0,1,0,1,1,1,0,1,0,1 ,
        1,0,0,1,1,0,1,1,0,0,1,0,0,0,0,1 ,
        1,1,0,0,1,0,0,1,1,0,1,0,1,0,0,1 ,
        1,0,0,1,1,1,1,0,0,0,0,0,1,1,0,1 ,
        1,0,0,0,1,0,0,1,1,0,1,1,1,0,0,1 ,
        1,1,0,0,0,1,0,1,0,0,0,0,0,0,0,1 ,
        1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1
    ];

    package def width:Integer = 16;
    package def height:Integer = 16;

    package def startX:Integer = 1;
    package def startY:Integer = 1;
    package def endX:Integer = 14;
    package def endY:Integer = 1;

    public-read var x:Integer;
    public-read var y:Integer;
    public-read var dir:Integer;
    public-read var success:Boolean =
        bind ((x==endX) and (y==endY));
// Part 2 is in listing 10.6
```

Maze wall data; 1 is a wall, 0 is an empty space

Dimensions of map

Start and end cells

Current player position/direction

Reached the end yet?

In listing 10.5 have the first half of our `Map` class. It begins with a huge table defining which parts of the map are walls and which are passageways. Immediately following that we have the dimensions of the map, 16 x 16 cells, followed by the cells that denote the start and end locations.

The `public-read` variables expose the current cell (x and y) the player is standing on and which direction the player is facing. North is 0, east is 1, south is 2, and west is 3. Finally we have a convenience boolean, `true` when the player's location is the same as the end location (and the maze has therefore been solved).

Let's move on the listing 10.6, which is the second part of the `Map` class.

Listing 10.6 Map.fx (part 2)

```
// Part 1 is listing 10.5
    init {
        x = startX;
        y = startY;
    }

    package function isEmpty(x:Integer,y:Integer) : Boolean {
        if(x<0 or y<0 or x>=width or y>=height) { return false; }
        var idx:Integer = y*width+x;
        return( wallMap[idx]==0 );
    }

    package function moveRel(rx:Integer,ry:Integer,rt:Integer) : Boolean {
        if(rx!=0 or ry!=0)
        {   def rot:Integer[] = rotateToView(rx,ry);
            if(isEmpty(x+rot[0],y+rot[1])) {
                x+=rot[0];  y+=rot[1];
                return true;
            }
            else {
                return false;
            }
        }
        else if(rt<0) {
            dir=(dir+4-1) mod 4;
            return true;
        }
        else if(rt>0) {
            dir=(dir+1) mod 4;
            return true;
        }
        else {
            return false;
        }
    }

    package function rotateToView(x:Integer,y:Integer) : Integer[] {
        [   if(dir==1) 0-y
                else if(dir==2) 0-x
                else if(dir==3) y
                else x ,
            if(dir==1) x
                else if(dir==2) 0-y
                else if(dir==3) 0-x
                else y
        ];
    }
}
```

Moving x/y?

Rotate coordinates

If possible, move to cell

Turn left (counterclockwise)

Turn right (clockwise)

Calculate absolute x coordinate

Calculate absolute y coordinate

In the conclusion of the `Map` class we have a set of useful functions for querying the map data and updating the player's position.

- The `init` block assigns the player's current location from the map's start location.
- `isEmpty()` returns `true` if the cell at `x` and `y` has no wall. We saw it in action during the update of the 3D maze, determining whether nodes should be visible or invisible.
- `moveRel()` accepts three parameters, controlling x movement, y movement, and rotation. It returns `true` if the move was performed. The parameters are used to move the player relative to the current position *and orientation* or rotate their view. If `rx` or `ry` is not 0, the relative positions are added to the current player location (the `rx` and `ry` coordinates are based on the direction the player is currently facing, so the `rotateToView()` orientates them the same way as the map). If the `rt` parameter is not 0, it is used to rotate the player's current orientation.
- `rotateToView()` is the function we've seen used repeatedly whenever we needed to translate relative coordinates facing in the player's current orientation to relative coordinates orientated north. If I move forward one cell, the relative movement is x = 0, y = -1 (vertically one cell up, horizontally no cells). But if I'm facing east at the time, this needs to be translated to x = 1, y = 0 (horizontally right one cell, vertically no cells) to make sense on the map data. The `rotateToView()` function translates this player-centric movement into a map-centric movement, returning a sequence of two values: x and y.

So that's our `Map` class. There are three simple custom nodes we need to look at before we pull everything together into our main application. So let's deal with them quickly.

10.2.3 The Radar class: this is where we are

The `Radar` class is a simple class for displaying the position of the player within the bounds of the maze, as show in the bottom-left corner of figure 10.7.

The radar has a kind of 8-bit retro style, in keeping with the simple, unfussy graphics of the maze. It doesn't show the walls of the maze—that would make the game too easy—just a pulsing circle representing where the player is and a yellow square for the goal.

Let's take a look at the code, courtesy of listing 10.7.

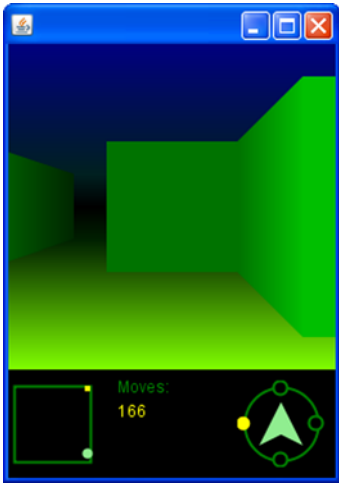

Figure 10.7 The maze game, complete with radar in the bottom left-hand corner and a compass in the bottom right

Listing 10.7 Radar.fx

```
package jfxia.chapter10;

import javafx.animation.transition.ScaleTransition;
import javafx.animation.Timeline;
import javafx.scene.CustomNode;
import javafx.scene.Group;
import javafx.scene.Node;
```

```
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.scene.shape.Rectangle;

package class Radar extends CustomNode
{   def cellSize:Number = 4;
    def border:Number = 8;
    public-init var map:Map;

    override function create() : Node {
        var c:Circle;
        var n:Group = Group {
            def w = map.width*cellSize;
            def h = map.height*cellSize;

            layoutX: border;
            layoutY: border;
            content: [
                Rectangle  {                          <-- Background rectangle
                    width: w;  height: h;
                    fill: null;
                    stroke: Color.GREEN;
                    strokeWidth:2;
                } ,
                c = Circle {                          <-- Circle presenting player
                    layoutX: cellSize/2;
                    layoutY: cellSize/2;
                    centerX: bind map.x * cellSize;
                    centerY: bind map.y * cellSize;
                    radius: cellSize;
                    fill: Color.LIGHTGREEN;
                } ,
                Rectangle {                           <-- End marker
                    x: map.endX * cellSize;
                    y: map.endY * cellSize;
                    width: cellSize;
                    height: cellSize;
                    fill: Color.YELLOW;
                }
            ];
            clip: Rectangle {
                width: w + border*2;
                height: h + border*2;
            }
        }

        ScaleTransition {                             <-- Infinite scale in/out
            node: c;
            duration: 0.5s;
            fromX: 0.2;  fromY: 0.2;
            toX: 1;  toY: 1;
            autoReverse: true;
            repeatCount: Timeline.INDEFINITE;
        }.play();

        n;
    }
}
```

The `Radar` class is a very simple scene graph coupled with a `ScaleTransition`. The `cellSize` is the pixel dimension each maze cell will be scaled to on our display. You'll recall from the `Map` class that the maze is 16 x 16 cells in size. At a pixel size of 4, this means the radar will be 64 x 64 pixels. And speaking of the `Map` class, the `map` variable is a local reference to the game's state and data. The `border` is the gap around the edge of the radar, bringing the total size to 80 x 80 pixels.

The scene graph manufactured in `create()` is very basic, but then it doesn't need to be fancy. Three shapes are contained within a `Group`, shifted by the `border`.

A background `Rectangle`, provides the boundary of the maze area, sized using the `map` dimensions against the `cellSize` constant. A `Circle` is used to show where the player currently is; its position is bound to the `cellSize` multiplied by the player's location in the `map`. Because a JavaFX `Circle` node's origin is its center point, we shift it by half a cell, to put the origin of the circle in the center of the cell. The end point in the map (the winning cell) is displayed using a yellow `Rectangle`.

Before we return the `Group` we create a `ScaleTransition`, continually growing and shrinking the circle from full size to just 20%. With the `autoReverse` flag set to `true`, the animation will play forward and then backward. And with `repeatCount` set to `Timeline.INDEFINITE`, it will continue to run, forward and backward, forever. (Well, at least until the program exits!)

10.2.4 The Compass class: this is where we're facing

The `Compass` class is another simple, retro-style class. This one spins to point in the direction the player is facing. Listing 10.8 is our compass code. It's a two-part custom node, with a static part that does not move and a mobile part that rotates to show the player's orientation. You can see what it looks like by glancing back at figure 10.7.

Listing 10.8 Compass.fx

```
package jfxia.chapter10;

import javafx.animation.transition.RotateTransition;
import javafx.scene.CustomNode;
import javafx.scene.Group;
import javafx.scene.Node;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.scene.shape.Polygon;
import javafx.scene.shape.Rectangle;

package class Compass extends CustomNode {
    def size:Number = 64;
    def blobRadius:Number = 5;

    public-init var map:Map;

    var compassNode:Node;

    override function create() : Node {
        def sz2:Number = size/2;
        def sz4:Number = size/4;
```

```
        compassNode = Group {    <-- Rotating group
            content: [
                Circle {                          <-- North circle (yellow)
                    centerX: sz2;
                    centerY: blobRadius;
                    radius: blobRadius;
                    fill: Color.YELLOW;
                } ,
                _makeCircle(blobRadius , sz2) ,
                _makeCircle(size-blobRadius-1 , sz2) ,
                _makeCircle(sz2 , size-blobRadius-1)
            ]                                     <-- West, east, and south circles (hollow)
            rotate: map.dir * 90;    <-- Initial rotation
        };

        Group {        <-- Static group
            content: [
                Circle {
                    centerX: sz2;  centerY: sz2;
                    radius: sz2-blobRadius;
                    stroke: Color.GREEN;
                    strokeWidth: 2;
                    fill: null;
                } ,                               <-- Ring circle

                Polygon {
                    points: [
                        sz2 , sz4 ,
                        size-sz4 , size-sz4,
                        sz2 ,  sz2+sz4/2 ,
                        sz4 , size-sz4
                    ];                            <-- Central arrow
                    fill: Color.LIGHTGREEN;
                } ,
                compassNode                       <-- Add in rotating group
            ]
            clip: Rectangle { width:size;  height:size; }
        }
    }

    package function update() : Void {    <-- Change rotation
        RotateTransition {
            node: compassNode;
            duration: 1s;
            toAngle: map.dir * 90;    <-- Animate to new direction
        }.play();
    }

    function _makeCircle(x:Number,y:Number) : Circle {    <-- Convenience function: make a circle
        Circle {
            centerX: x;  centerY: y;
            radius: blobRadius;
            stroke: Color.GREEN;
            strokeWidth: 2;
        }
    }
}
```

The instance variable `size` is the diameter of the ring, while `blobRadius` helps size the circles around the ring. (Blob? Well, can *you* think of a better name?)

Inside `create()`, the `compassNode` group is the rotating part of the graph. It is nothing more than four circles (blobs) representing the points of the compass, with the northern position a solid yellow color. The other three are hollow and identical enough to be created by a convenience function, `_makeCircle()`. The `compassNode` is plugged into a static part of the scene graph, comprising a ring and an arrow polygon.

The `update()` function is called whenever the direction the player is facing changes. It kicks off a `RotateTransition` to spin the blob group, making the yellow blob face the player's direction. Because we specify only an end angle and no starting angle, the transition (conveniently) starts from the current rotate position.

10.2.5 The ScoreBoard class: are we there yet?

Only one more custom node to go; then we can write the application class itself. The scoreboard keeps track of the score and displays a winning message; its code is in listing 10.9.

Listing 10.9 ScoreBoard.fx

```
package jfxia.chapter10;

import javafx.scene.CustomNode;
import javafx.scene.Node;
import javafx.scene.layout.VBox;
import javafx.scene.paint.Color;
import javafx.scene.text.Text;
import javafx.scene.text.TextOrigin;

package class ScoreBoard extends CustomNode {
    public-init var score:Integer;
    package var success:Boolean = false;

    override function create() : Node {
        VBox {
            spacing: 5;
            content: [
                Text {
                    content: "Moves:";
                    textOrigin: TextOrigin.TOP;
                    fill: Color.GREEN;
                } ,
                Text {
                    content: bind "{score}";
                    textOrigin: TextOrigin.TOP;
                    fill: Color.YELLOW;
                } ,
                Text {
                    content: bind if(success)
                        "SUCCESS!" else "";
                    textOrigin: TextOrigin.TOP;
                    fill: Color.YELLOW;
                }
```

Static text: "Moves:"

Dynamic text: score

Success message

```
            ];
        };
    }

    package function increment() : Void
    {   if(not success)  score++;
    }
}
```

Increment score, if not success

The score panel is used in the lower-middle part of the game display. It shows the moves taken and a message if the player manages to reach the winning map cell. You can see it in action in figure 10.8.

There's not a lot to mention about this bit of scene graph; it's just three `Text` nodes stacked vertically. The `increment()` function will add to the score, but only if the `success` flag has not been set. As you'll see when we check out the application class, `success` is set to `true` once the end of the maze is reached and never unset. This prevents the score from rising once the maze has been solved.

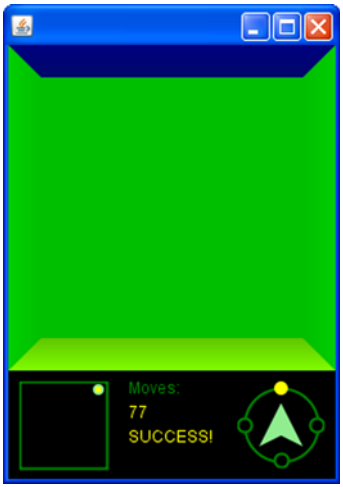

Figure 10.8 The scoreboard sits at the bottom of the display, showing the moves used and a "SUCCESS!" message once the end of the maze is reached.

10.2.6 The MazeGame class: our application

At last we get to our application's main class, where we pull the four custom nodes previously discussed together into our game's UI. This is our application class (listing 10.10), and thanks to all the work we did with the custom nodes, it looks pretty compact and simple.

Listing 10.10 MazeGame.fx

```
package jfxia.chapter10;

import javafx.scene.Scene;
import javafx.scene.Group;
import javafx.scene.input.KeyCode;
import javafx.scene.input.KeyEvent;
import javafx.stage.Stage;

def map:Map = Map{};
var mapDisp:MapDisplay;
var scBoard:ScoreBoard;
var comp:Compass;
var gp:Group;

Stage {
    scene: Scene {
        content: gp = Group {
            content: [
```

```
                mapDisp = MapDisplay {
                    map: map;
                } ,
                Radar
                {   map: map;
                    layoutY: 240;
                } ,
                scBoard = ScoreBoard {
                    layoutX: 88;
                    layoutY: 248;
                } ,
                comp = Compass {
                    map: map;
                    layoutX: 168;
                    layoutY: 248;
                }
            ];
            onKeyPressed: keyHandler;
        }
        width: 240;  height: 320;
        fill: javafx.scene.paint.Color.BLACK;
    }
}
gp.requestFocus();

function keyHandler(ev:KeyEvent) : Void {
    def c = ev.code;
    def x:Integer =
        if(c==KeyCode.VK_LEFT) -1
        else if(c==KeyCode.VK_RIGHT) 1
        else 0;
    def y:Integer =
        if(c==KeyCode.VK_UP) -1
        else if(c==KeyCode.VK_DOWN) 1
        else 0;
    def t:Integer =
        if(c==KeyCode.VK_SOFTKEY_0 or
            c==KeyCode.VK_OPEN_BRACKET) -1
        else if(c==KeyCode.VK_SOFTKEY_1 or
            c==KeyCode.VK_CLOSE_BRACKET) 1
        else 0;

    if(x==0 and y==0 and t==0)  return;

    if( map.moveRel(x,y,t) ) {
        mapDisp.update();  comp.update();
        if(t==0)  scBoard.increment();
    }

    if(map.success)  scBoard.success=true;
}
```

3D map display

Radar, lower left

ScoreBoard, lower middle

Compass, lower right

Install keyboard handler

Default background is white

Request keyboard focus

Which key was pressed?

Move left or right?

Move forward or backward?

Turn left or right?

Nothing to do? Exit!

Perform movement or turn

Have we won?

After defining a few variables, we move straight into the scene graph. Since this is a top-level application class, we use a `Stage` and a `Scene` and then plug our custom nodes into it via a `Group`. The `MazeDisplay` is added first, followed by the `Radar` positioned into the bottom left-hand corner. Next to the radar is the `ScoreBoard`, and finally the `Compass` takes up the bottom right-hand corner.

The reason we used a `Group`, rather than plug the nodes directly into the `Scene`, is to give us a node we can assign a keyboard handler to. In this case it's a function by the name of `keyHandler`, defined outside the graph for readability sake. To make the handler work we need to request the keyboard input focus, which is the purpose of the `gp.requestFocus()` call.

What of the handler itself? The `KeyEvent` parameter contains all manner of information about the state of the keyboard when the event was generated. In this case we just want the raw code of the key that got pressed, which we copy into a handy variable. We compare this variable against the constants in the `KeyCode` class, to populate three more variables: `x`, `y` and `t`, such that `x` is -1 if moving left and 1 if moving right, `y` is -1 if moving forward and 1 if moving backward, and `t` is -1 if turning left (counterclockwise) and 1 if turning right (clockwise).

You'll recognize these values as the relative movements the `Map` class accepts to make a movement, which is precisely what we do using `map.moveRel()`. The function returns `true` if the player successfully moved or turned, which causes us to update the `MazeDisplay` and the `Compass`. If the action was a move (not a turn), we also increment the moves counter in the `ScoreBoard` class.

Finally we check to see if the winning cell has been reached and set the `ScoreBoard`'s `success` flag if it has. Note that it never gets unset; this is so the score won't rise if the player continues to move once the maze has been solved.

And that's our complete game. Now let's try it out.

10.2.7 Running the MazeGame project

We can run the game in the usual way and navigate around in glorious `3D` using the keyboard's cursor keys (the arrow keys) and the square bracket keys for turning. Without cheating by looking at the data, see if you can navigate your way to the end using only the radar and compass to aid you.

Take a look back at figures 10.7 and 10.8 to see how the game looks in action.

The game works perfectly well on the desktop, but our ultimate goal is to transfer it onto a mobile device emulator. If you're wondering how much we'll have to change to achieve that aim, then the next section might come as a pleasant surprise.

10.3 On the move: desktop to mobile in a single bound

It's time to move our application onto a cell phone or, more specifically, an emulator that simulates the limited environment of a cell phone (see figure 10.9).

In the source code for the `MazeGame` class (listing 10.10) you may have noticed each action of turning, either clockwise or counterclockwise, was wired to a couple of keys. The `KeyCode.VK_SOFTKEY_0` and the `KeyCode.VK_SOFTKEY_1` could be used in preference to `KeyCode.VK_OPEN_BRACKET` and `KeyCode.VK_CLOSE_BRACKET`. As you may have guessed, this is to accommodate the limited key input on many mobile devices.

This is really the only concession I had to make for the mobile environment. The rest of the code is mobile ready. When writing the game I was very careful to use only those classes available for the *common profile*.

The common what?

The term *profile* is used to refer to given groupings of a JavaFX class. Not every class in the JavaFX API is available across every device, because not every device has the same power or abilities. Classes that can be used only on the desktop, for example, are in the *desktop profile*. Similarly classes that may be available only on JavaFX TV are exclusive to the *TV profile* (hypothetically speaking, given that the TV platform is months from release as I write). Classes that span all environments, devices, and platforms are said to be in the *common profile*. By sticking to only the part of the JavaFX API that is in the common profile, we ensure that the maze game will work on all JavaFX platforms, including the phone emulator. But how can we find out which classes are supported by which profile?

The official JavaFX web documentation features a toggle, in the form of a group of links at the head of each page, for showing only those bits of code inside a given profile. This is handy when we wish to know the devices and platforms our finished code will be able to run on. Flip the switch to "common," and all the desktop-only parts of the API will vanish from the page, leaving only those safe to use for mobile development.

Figure 10.9 Our maze game hits the small screen. More specifically, it's running on the JavaFX 1.2 mobile emulator.

So, we don't need to make any changes to the software; our game is already primed to go mobile. We just need to know how to get it there!

10.3.1 Packaging the game for the mobile profile

To get the game ready for the emulator we use the trusty `javafxpackager` tool, first encountered in the Enigma chapter, to bundle code ready for the web. In this scenario, however, we want to output a MIDlet rather than an applet (MID stands for Mobile Information Device). MIDlets are Java applications packaged for a mobile environment, and here's how we create one:

```
javafxpackager -profile MOBILE -src .\src
  -appClass jfxia.chapter10.MazeGame
  -appWidth 240 -appHeight 320
```

The example is the command line we need to call the packager with, split over several lines. If you're on a Unix-flavored machine, the directory slashes run the opposite way, like so:

```
javafxpackager -profile MOBILE -src ./src
  -appClass jfxia.chapter10.MazeGame
  -appWidth 240 -appHeight 320
```

The command assumes you are sitting in the project's directory, with `src` immediately off from your current directory (and the JavaFX command-line tools on your path). The options for the packager are quite straightforward. The application's source directory, its main class name, and its dimensions should all be familiar. The only change from when we used the packager to create an applet is the `-profile MOBILE` option. This time we're asking to output to the mobile environment instead of the desktop. (If you need to remind yourself of how the packager works, take a look back at section 9.4.2, in the previous chapter.)

A dist directory is created within our project, and inside you should find two files:

- The MazeGame.jar file is the game code, as you would expect.
- Its companion, MazeGame.jad, is a definition file full of metadata, helping mobile devices to download the code and users to know what it does before they download it.

> **Going mobile, with NetBeans**
>
> Once again I'm showing you how things happen under the hood, without the abstraction of any given integrated development environment (IDE). If you're running NetBeans, you should be able to build and package the mobile application without leaving your development environment. Terrence Barr has written an interesting blog entry explaining how to create and build mobile projects from NetBeans (the following web address has been broken over two lines).
>
> http://weblogs.java.net/blog/terrencebarr/archive/2008/12/javafx_10_is_he.html

The JAR and the JAD file are all we need to get our application onto a phone. The JAR is the code and the JAD is a property file with metadata for the application. But before our software goes anywhere near a real phone, we'd obviously want to test it in a development environment first. The next step, therefore, is to learn how to fire up the mobile emulator.

10.3.2 Running the mobile emulator

The mobile emulator is a simulation of the hardware and software environment in typical mobile phones. The emulator enables us to test and debug a new application without using an actual cell phone.

You'll find the emulator inside the emulator\bin directory of your JavaFX installation. If you're planning to run the emulator from the command line, check to make sure that this directory is on your execution search path. To run the emulator with our maze game, we merely need to point its `-Xdescriptor` option at our JAD file, like so:

```
emulator -Xdescriptor:dist\MazeGame.jad
```

When you run this command, a little service program will start up called the JavaFX SDK 1.2 Device Manager. You'll see its icon in the system tray (typically located in the southeast corner of the screen on Windows). If you are running a firewall (and I sincerely hope you are), you may get a few alerts at this point, asking you to sanction connections from this new software. Providing the details relate to Java and not some other mysterious application, you should grant the required permissions to allow the Device Manager and emulator to run. The result should look like figure 10.10.

Once the emulator has fired up, the game can be played using the navigation buttons and the two function buttons directly below the screen. The navigation buttons move around the maze, while the function buttons turn the view.

Running the mobile emulator, with NetBeans

If you want to remain within your IDE while testing mobile applications, there's a guide to working with the emulator from NetBeans under the section titled "Using the Run in Mobile Emulator Execution Model" at the following web address:

http://javafx.com/docs/tutorials/deployment/configure-for-deploy.jsp

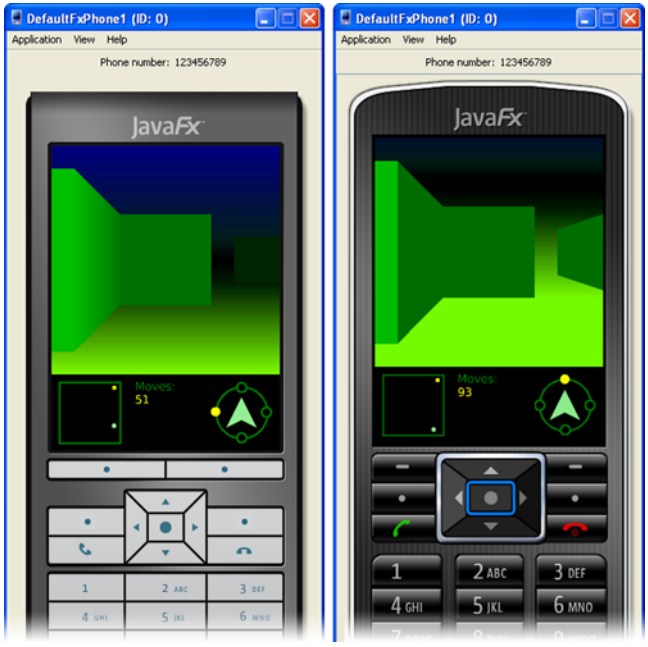

Figure 10.10 The old JavaFX 1.1 mobile emulator (left) and its 1.2 update (right) in action. Strangely, the older version seems to reproduce the gradient paints better.

And that's it; our maze has successfully gone mobile!

10.3.3 Emulator options

The emulator has a range of command-line options, outlined in table 10.1. Options have been grouped by association.

Table 10.1 Emulator options

Options	Function
`-version` `-help`	Print version or help information. Use the latter to get more information on other options.
`-Xdebug` `-Xrunjdwp`	Allow a debugger to connect to the emulator and optionally set a Java Wire Debug Protocol (JWDP) for the emulator/debugger to use when communicating.
`-Xquery` `-Xdevice`	List available devices the emulator can simulate along with their attributes, or specify which device to employ when running software on the emulator.
`-Xdescriptor` `-Xautotest`	Run a given mobile application via its JAD file, or automatically test each mobile application in a MIDlet suite (in theory it's possible to package several mobile applications into one JAR, although this feature and the associated autotest option are rarely used in practice).
`-Xjam`	Send commands to the Java Application Manager (JAM) simulator on the emulator. The JAM manages the phone's installed applications and permits OTA (over the air) installation from a URL.

Several of these options accept parameters; consult the emulator's local documentation page (it came with the rest of the documentation when you installed JFX on your computer) or use the `-help` switch to get a list of what is required for each option.

Generally you'll be using `-Xdescriptor` to test your software, perhaps with `-Xdebug` to allow a remote debugger to be used as your code runs. The `-Xquery` and `-Xdevice` options can be used to test your application on differing mobile device configurations. These configurations do not necessarily re-create specific models of a real-world phone but rather generalized approximations of devices available on the market. The `-Xjam` option allows you to re-create a true mobile application lifecycle, from over the air deployment to eventual deinstallation.

Next up is testing our application on a real phone.

10.3.4 Running the software on a real phone

We've come this far, and no doubt you're now eager to learn how to get the software onto a real phone. If so, I'm afraid this section might be a little disappointing.

As this chapter is being written, JavaFX Mobile is too new for there to be many physical devices readily available to test our application on. At JavaOne 2009 Sun made available some HTC Touch Diamond phones running JavaFX 1.2. These were intended as beta release hardware for developers to test their applications on. Consumer-focused JavaFX Mobile phones are being worked on, but right now it's not possible to walk into

More on Java ME

Judging by the emulator shipped with JavaFX SDK v1.2, the JFX mobile emulator owes a lot to recent developments in Java ME (Micro Edition). The first link (broken over two lines) points to detailed information on the new Java ME emulator and its options, while the second is a portal page to Java ME's latest developments:

http://java.sun.com/javame/reference/docs/sjwc-2.2/pdf-html/html/tools/index.html

http://java.sun.com/javame/index.jsp

For more background on how Java ME MIDlets interact with a phone and its OS (including its lifecycle), plus details of the JAD file format, consult these links:

http://developers.sun.com/mobility/learn/midp/lifecycle/

http://en.wikipedia.org/wiki/JAD_(file_format)

a store and come out with a phone that runs JavaFX out of the box or install an upgrade to support it.

Assuming JavaFX follows the same basic deployment scheme as the current Java Micro Edition, we can make an educated guess as to how it might work once JavaFX starts to become available on actual consumer devices. By uploading both the JAR and JAD onto a web server, we can make them accessible for public download. Pointing the phone toward the JAD's URL (the file may need editing first, to provide an absolute URL to its companion JAR) should cause the phone to query the user about downloading the application. If the user accepts, the software will be transferred onto the phone and made available on its application menus.

Obviously, the process may vary slightly from phone to phone and OS to OS, but the general theory should hold true across all devices: the JAD file is the primary destination, which then references the actual application JAR file.

Who's on board?

So where is JavaFX Mobile going in terms of real-world devices? Who's on board from the handset manufacturers and other hardware providers? The following web links collect documentation and press releases detailing prominent hardware partners Sun has announced at the time of writing:

http://javafx.com/partners/

http://www.sun.com/aboutsun/pr/2009-02/sunflash.20090212.1.xml

http://developer.sonyericsson.com/site/global/newsandevents/latestnews/newsfeb09/p_javafxmobile_sonyericsson_announcement.jsp

It's a shame we can't currently run our applications on real-world phones, but the emulator does a good job of preparing us for the near future, when we should be able to. It allows us to see not only how the interface will work but also how the performance will rank against the desktop. Efficient performance is key to writing good mobile software, so it's worth spending our final section before the summary looking at a few simple optimization techniques.

10.4 Performance tips

When developing on the mobile platform it's important to pay particular attention to how expensive various operations are. Things a desktop computer can do without breaking a sweat may really tax smaller devices. The following are notes on how to keep your mobile applications zipping along at top speed. This list is by no means exhaustive. It's compiled from my own observations with SDK 1.2, comments from other JavaFX coders, and advice from Sun engineers.

- Binds are useful but carry an overhead. While you shouldn't avoid them altogether, you need to be sensible in their use. Too many bound variables can cause significant slowdowns in your application. Avoid binding to data that is unlikely to change during the lifetime of the application, for example, the screen resolution. When variables need to respond to changes in other variables, consider using triggers on the source to *push* updates out rather than binds on the recipients to *pull* them in.
- Image manipulation can be expensive, particularly if the images are large. Setting the `width` and `height` of an `Image` will cause it to be prescaled to the required size as it loads. Scaling images to fit a display is obviously more common on mobile applications; valuable resources can be saved by doing it up front rather than continually during live scene graph updates.
- In a mobile environment, where the UI might need to be resized to fit the device's resolution, a shapes-based approach is often better than an image-based approach. It's a lot less expensive to resize or transform shapes than images, and the results look cleaner too. However, see the next point.
- Oversized scene graphs can cause headaches. The precise problems will likely vary from device to device, depending on the graphics hardware, but in general scene graphs should be culled of unnecessary shapes, invisible nodes, or fully transparent sections when possible. Rather than hiding unused parts of a user interface, consider adding/removing them from the stage's *scene* as required. Sometimes *fixing* your UI into a bitmap image is better than forcing JavaFX Mobile to render a complex scene graph structure with every update. (Shapes versus images is something of a fine balancing act.)
- As a follow-up to the previous point, be aware that text nodes, because of their nature, are complex shapes. Devices may have efficient font-rendering functions built into their OS, but these may not be so hot for drawing transformed text.

Credit where credit's due

You can find these tips, plus more (including examples), in an article by Michael Heinrichs titled "JavaFX Mobile Applications — Best Practices for Improving Performance." The web address has been broken over two lines:

http://java.sun.com/developer/technicalArticles/javafx/mobile/index.html

You'll note in the maze game that I broke one of these rules. I switched the visibility of wall nodes rather than remove unwanted nodes from the scene graph. The code was tough enough, I felt, without confusing readers with more complexity. The maze game seems to work fine on the emulator, but to squeeze out maximum performance we should ditch the `wallVisible` sequence and change the `update()` function to rebuild the scene graph from scratch with each move, including only visible nodes.

10.5 Summary

In this chapter we've pushed the scene graph further than anything we've seen before, traveling beyond the flat world of two dimensions. We also got a small taste of JavaFX Mobile, although only via the emulator for now.

As always with the projects in this book, there's still plenty of room for experimentation. For example, the game works fine on 240 x 320 resolution displays but needs to be scaled and rearranged for alternative screen sizes. (Hint: the `MazeDisplay` class supports a scaling factor, which you might want to investigate.)

This chapter went through quite a journey during the writing of the book, some of which is worth mentioning from a technical point of view. It was earmarked originally as a mobile chapter, but by the fall of 2008 it looked increasingly uncertain whether mobile support would be in the initial JavaFX release. The first draft was targeted at the desktop, and featured a far more complex scene graph that applied perspective effects onto bitmap images, overlaid with translucent polygons to provide darkness. Very *Dungeon Master* (see figure 10.11). However, when JavaFX 1.0 arrived in December 2008, complete with bonus Mobile preview, the code was stripped back radically to become more suitable for a mobile device.

Originally I wanted to manufacture the 3D scene using SVG: create a vector-based wall image and replicate it for all the sizes and distortions needed to form the 19 wall pieces. Each piece would be on a different (`jfx:` labeled) layer inside a single SVG and could be switched on or off independently to show or hide the wall nodes. A little bit of extra effort up front with Inkscape (or Illustrator) would produce much cleaner source code, effectively removing the need for the `_wallFront()` and `_wallSides()` functions. Imagine my disappointment when I discovered the FXD library wasn't yet compatible with the JavaFX 1.0 Mobile profile.

Figure 10.11 A desktop version of the 3D maze, complete with bitmap walls using a perspective effect. Sadly, the bitmaps had to go when the project was adapted to fit a mobile platform.

So it's early days for the Mobile profile, clearly, but even the debut release used for this chapter shows great promise. As mobile devices jump in performance, and their multimedia prowess increases, the desktop and the handheld spaces are slowly converging. JavaFX allows us to take advantage of this trend, targeting multiple environments in a single bound through a common profile, rather than coding from scratch for every platform our application is delivered on.

So far in the book we've used JavaFX on the desktop, taken a short hop over to a web browser, then a mighty leap onto a phone, all without breaking a sweat. Is there anywhere else JavaFX can go? Well, yes! It can also run *inside* other applications, executing scripts and creating bits of UI. Although a rather esoteric skill, it can (in very particular circumstances) be incredibly useful. So that's where we'll be heading next.