Flex 4 Fun

Excerpt

artima

ARTIMA PRESS
MOUNTAIN VIEW, CALIFORNIA

Buy the Book · Discuss

Thank you for downloading this sample eBook chapter from the First Edition of *Flex 4 Fun*. The only difference between the actual eBook chapter and this sample is that in this document, the page numbers start at 1, and references outside the chapter are not hyperlinked. In the actual eBook, those reference also serve as a hyperlink to the referenced page in the book.

The difference between this chapter and its appearance in the paper book is that the paper book uses shades of gray, not color, for figures and to syntax highlight code. We hope you find this sample chapter useful and enjoyable.

Bill Venners
President, Artima, Inc.

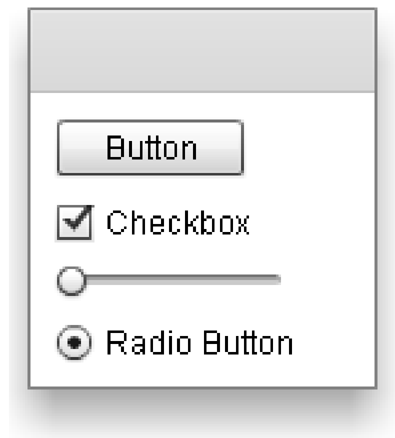
Copyright © 2010 Chet Haase. All rights reserved.

Chapter 2

Graphics

Graphics are the heart of GUI applications. Graphical objects are used to describe the visual appearance of components as well as to create custom rendering like gradient backgrounds. The richness of the graphics platform in a GUI toolkit determines how easily you can build rich client applications with that toolkit. Since Flex sits atop the Flash platform, there is a wealth of graphics capabilities available, enabling very rich clients indeed.

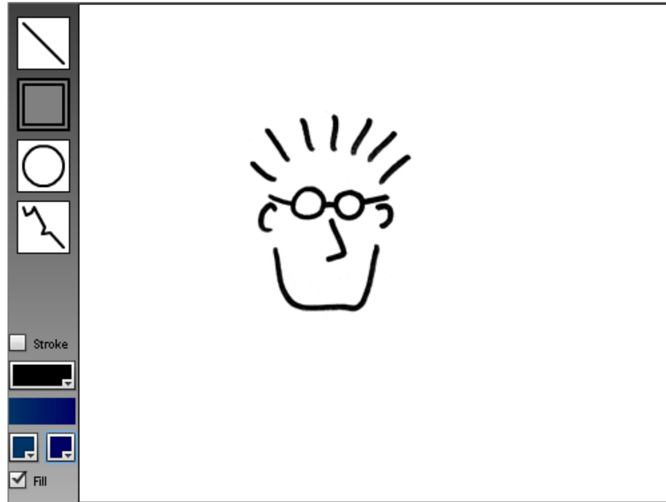
You can see the use of graphics objects in every Flex 4 component, like this panel full of controls:



Every one of these components is made up of simple graphics primitives. The panel consists of a couple of rectangles with a darker fill for its header. The button is a rounded rectangle filled with a light gradient and stroked

with a darker border color. The checkbox has a filled and stroked rectangle for the box and a path object for the check mark. The slider is composed of a rounded rectangle for the track and a circle for the thumb. And the radio button has one circle for the button and one for the selection indicator.

You use these same graphics primitives to create very custom and dynamic objects on the screen, as seen here:



(Demo: Shapely)

In this drawing application, all of the control panel objects on the left as well as the scribbled face on the drawing canvas were created with graphic elements. We'll see more of the `Shapely` application later in this chapter as we explore various shapes and drawing attributes available in Flex 4.

Flex 4 graphics

Once upon a time (as far back as Flex 3), if you wanted custom graphics, you had to dive into ActionScript code, override a method or two, create and use Flash display objects, and issue calls into their `Graphics` objects. It was the only way to draw custom graphics from your Flex application. For example, here's how you might draw a circle in Flex 3:

(File: ThreeCircles.mxml)

```
var component:UIComponent = new UIComponent();
var sprite:Sprite = new Sprite();
sprite.graphics.beginFill(0xff0000);
sprite.graphics.drawEllipse(0, 0, 100, 100);
sprite.graphics.endFill();
component.addChild(sprite);
addElement(component);
```

Flex 4 provides a new graphics API that allows you to easily create objects that describe visual elements. The Flex library internally handles the details of telling Flash how to create and render these objects. For example, here's a simple circle using the new graphics classes of Flex 4:

```
var circle:Ellipse = new Ellipse();
circle.width = 100;
circle.height = 100;
circle.fill = new SolidColor(0x0000ff);
circle.x = 100;
addElement(circle);
```

And here's a even better example of the Flex 4 approach, using some of the new MXML tags:

```
<s:Ellipse x="200" y="0" width="100" height="100">
    <s:fill>
        <s:SolidColor color="green"/>
    </s:fill>
</s:Ellipse>
```

You will notice some important differences between the old way of creating graphics and the new way of doing it in Flex 4:

Declarative The approach of creating graphics in Flex 4, like much of the rest of Flex, is object-oriented and declarative. You create the graphics primitive you need, set the properties of that object to tell it how to draw itself, and add it to the appropriate container in your application. The old way of drawing in Flex 3 was, by contrast, very manual. You got a reference to Flash `Graphics` object and called drawing functions on that object to tell the object how to render itself.

MXML Because the new graphic elements are declarative, you can use MXML markup to describe your visuals. You can now use MXML to describe the visual aspects of your program, and only dive into ActionScript for the more programmatic functionality of your application, like the business logic. This is particularly important when customizing the look of Flex 4 components through their *skins*. These component skins, which are written in MXML files, hold the graphical elements that describe the component's appearance. We will see more about component skinning in Chapter 6.

Flex-friendly You'll notice, in the Flex 3 code in the previous example where we draw into a sprite graphics object, an indirect approach of adding our sprite to a `UIComponent`, which is then added to our Flex application. This is because Flex 3 applications only understand components, not raw Flash display objects like our `Sprite` above. So whenever we want custom graphics in a Flex 3 application, they need to be drawn into a custom component, or added into a UIComponent, or by some other means added indirectly to the Flex display list. The Flex 4 approach is much more tightly integrated with Flex overall. We create `GraphicElement` objects, like the `Ellipse` in the Flex 4 code in the example, and add them directly to Flex containers.

For the rest of the chapter, we'll see the different kinds of graphics objects that we can create.

`Shapely`: a simple drawing tool

As you read through this chapter, learning about strokes and fills and the various graphic elements that you can use in Flex 4, you will see these objects in use in the `Shapely` drawing application seen earlier on page 3. But in order to understand how the various objects fit into that application, we'll first need to understand how the application works in general. Let's go over its basic architecture and mechanisms.

The `Shapely` application is a simple drawing application that allows the user to select between a small set of shapes (lines, rectangles, ellipses, and paths) along with different stroke and fill modes for the shapes. The user can then draw these shapes on the canvas with the mouse. In order to keep the application simple, both in terms of the UI and the code that we need to

digest, the application does not expose the full spectrum of graphic primitives and fill/stroke options that we cover in this chapter. But the application is a good place from which to start if you want to further enhance it with that additional functionality yourself.

Some of the code that you will see in the application uses techniques or functionality in Flex 4 that we have not yet covered. For example, some of the code in the `ControlPanel` class uses the new states syntax (which I talk about in Chapter 4) to change the look of the drawing primitive icons when the icons are selected. Don't worry too much about these bits of the code. The real point in this chapter is to understand how the application UI is drawn, using the graphics primitives and fills/strokes that we discuss here, and how the shapes are created by the user when dragging the mouse around. So let's get into the code.

First of all, take a look at the application GUI. As you can see in the screenshot on page 3, a control panel at the left contains buttons that the user selects to choose the current shape and the drawing options for that shape. The rest of the window contains a drawing canvas, where the user drags the mouse to draw shapes. The following sections will cover how the main application, the control panel, and the drawing canvas work.

The `Shapely` application class

The top-level application is very simple; it just instantiates and positions the custom components for the control panel and the drawing canvas:

(File: Shapely.mxml)

```
<components:ControlPanel id="controlPanel"
    width="52" height="100%"
    currentStateChange="drawingModeChange()"
    drawingStateChange="drawingStateChange(event)"/>
<components:DrawingCanvas id="canvas"
    left="52" right="0"
    top="0" bottom="0"/>
```

Some additional logic is in script code to handle setting the drawing state for the application. The drawing state is determined by actions in the control panel. A change in the shape to be drawn results, through the event mechanism, in a call to the `drawingModeChange()` function:

```
private function drawingModeChange():void
{
    switch (controlPanel.currentState)
    {
        case "lineMode":
            canvas.drawingMode = DrawingCanvas.LINE;
            break;
        case "rectMode":
            canvas.drawingMode = DrawingCanvas.RECT;
            break;
        case "ellipseMode":
            canvas.drawingMode = DrawingCanvas.ELLIPSE;
            break;
        case "pathMode":
            canvas.drawingMode = DrawingCanvas.PATH;
            break;
    }
}
```

A change in the stroke or fill controls sends a `drawingStateChange` event and results in a call to the `drawingStateChange()` function:

```
 private function drawingStateChange(
    event:DrawingStateChangeEvent):void
{
    canvas.stroke = event.stroke;
    canvas.fill = event.fill;
}
```

When the application receives these events, it sets the appropriate state in the drawing canvas to be used in future drawing operations. These events are received when the user interacts with the control panel.

The `Shapely` control panel

Now let's take a look at the control panel, from the file `ControlPanel.as`. This component is a subclass of `Group`, the simplest Flex 4 container and the base class for other container classes. `ControlPanel` is just a basic container for the various icons that control drawing state.

Each of the drawing primitive icons (the line, rectangle, ellipse, and path) have the same structure: a `Group` contains the icon graphics and listens for click events to signal that the user wants to switch to this drawing mode. For example, here is the group container for the line drawing icon:

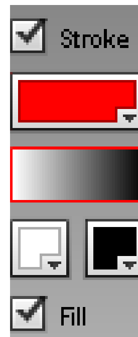
```
(File: components/ControlPanel.mxml)
<s:Group id="line" width="40" height="40"
    click="setMode(event)">
    <!-- group contents -->
</s:Group>
```

You can see in the previous code that a mouse click results in a call to the `setMode()` function. This function sets the drawing mode by setting the `currentState` of the component:

```
private function setMode(event:MouseEvent):void
{
    switch (event.currentTarget)
    {
        case line:
            currentState = "lineMode";
            break;
        case rect:
            currentState = "rectMode";
            break;
        case ellipse:
            currentState = "ellipseMode";
            break;
        case path:
            currentState = "pathMode";
            break;
    }
}
```

When any icon is clicked, it dispatches an event to this function, which sets the `currentState` property according to the button that was clicked. That change to `currentState` causes Flex to dispatch a `currentStateChanged` event, which is received by the `drawingModeChange()` function in `Shapely` that we saw earlier. This function sets the drawing mode for the canvas (which we will see later).

Finally, components at the bottom of the control panel determine the stroke and fill used when drawing:



Two checkboxes determine whether the object will be stroked and/or filled. In between the checkboxes are `ColorPicker` components for the stroke and the fill gradient (choose both fill colors to be the same to get a solid color). A preview rectangle between the stroke and fill sections shows the user what shapes will look like with the current stroke and fill settings. When any of these settings are changed, the `setDrawingState()` function is called. This function will be discussed later, after sections on the stroke and fill mechanisms for graphics primitives.

That's it for the control panel; onto the canvas, where the work is done for creating and drawing the shapes.

The `Shapely` drawing canvas

The main job of the `DrawingCanvas` class is handling mouse events and turning them into shapes on the canvas. It does this by listening first for `mouseDown` events, then to further `mouseMove` and `mouseUp` events to track mouse dragging actions by the user. The `mouseDown` listener is added to the canvas object in its constructor:

```
(File: components/DrawingCanvas.as)
addEventListener("mouseDown", mouseDownHandler);
```

When the user drags the mouse around the canvas, the code creates and manipulates different shapes according to the drawing mode that the user selected in the control panel. The following constants and variables are used to track the shape that is created on `mouseDown`:

```
public static const LINE:int = 0;
public static const RECT:int = 1;
public static const ELLIPSE:int = 2;
public static const PATH:int = 3;

public var drawingMode:int = LINE;
```

The drawingMode property is set by the application code in Shapely in its drawingModeChange() function, as we saw earlier. This property is used when we handle mouseDown events in our mouseDownHandler() function.

A mouseDown event on the canvas results in an event which causes a call to the mouseDownHandler() function. This function creates a new shape (which we'll see later, when we discuss shapes) and adds listeners for both mouseMove and mouseUp events:

```
addEventListener(MouseEvent.MOUSE_MOVE, mouseMoveHandler);
addEventListener(MouseEvent.MOUSE_UP, mouseUpHandler);
```

Note that the application only bothers listening for mouseMove and mouseUp events after receiving an initial mouseDown event. If the user is simply moving the mouse around without pressing it first, then it does not matter because they are not drawing a shape. But as soon as the user presses the mouse button, a shape is created and mouse movement is tracked to allow editing the shape with drag operations.

A mouse drag causes a call into the mouseMoveHandler() function:

```
private function mouseMoveHandler(event:MouseEvent):void
{
    dragTo(event.localX, event.localY);
}
```

This function calls the dragTo() function to change the shape currently being drawn, according to the current location to which the user has dragged the mouse. When the user releases the mouse button, there is a call to the mouseUpHandler() function:

```
private function mouseUpHandler(event:MouseEvent):void
{
    dragTo(event.localX, event.localY);
    removeEventListener(MouseEvent.MOUSE_MOVE,
        mouseMoveHandler);
```

```
        removeEventListener(MouseEvent.MOUSE_UP,
            mouseUpHandler);
}
```

As in the mouseMoveHandler() function, we call dragTo() to change the shape according to this final mouse location. We then remove our move/up listeners because we no longer care about these events until the next time the user presses the mouse button down.

That's it for the main application functionality of Shapely. The rest of the application code is about the shapes that are created and the drawing attributes that those shapes have. We'll see how all of these work as we cover these topics in the rest of this chapter.

Graphics primitives: getting into shape

Your first reaction to learning about the graphic elements in Flex might have been: "What can I draw?" You may think, given that all of the Flex components are drawn with these shapes, there would be a myriad of different shapes to choose from. But in fact, there is just a small set: Line, Rect, Ellipse, and Path. With just these four shapes, and with the stroke and fill options that we'll discuss later, you can draw all kinds of things, from simple graphics for components, like lines, circles, and rounded rectangles, to very custom artwork.

In this section, we'll see how each of these shapes are created and used in the Shapely application.

The Line class

Lines are the simplest graphics primitive; they are just single-segment connectors between endpoints. You can change what the lines look like using the stroke properties that we'll see in Section 1, but the basic geometry of lines is very simple: they start at one point and end at another.

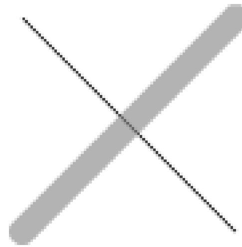
The Line class defines simple endpoint properties for a single line segment. These two endpoints are described by the (xFrom, yFrom) and (xTo, yTo) properties. Here are two sample lines created in MXML code:

(File: SimpleObjects.mxml)

```
<s:Line xFrom="20" yFrom="20" xTo="100" yTo="100">
```

```
    <s:stroke>
        <s:SolidColorStroke color="black"/>
    </s:stroke>
</s:Line>
<s:Line xFrom="30" yFrom="20" xTo="110" yTo="100">
    <s:stroke>
        <s:SolidColorStroke color="gray" alpha=".8"
            weight="5"/>
    </s:stroke>
</s:Line>
```

This MXML code results in the following graphics on the screen:



(Demo: SimpleObjects)

Don't worry about the `stroke` objects in the code yet; we'll read more about strokes in Section 1.

Now let's see how we create `Line` objects in the `Shapely` application. When the user presses the mouse key down on the drawing canvas, the function `mouseDownHandler()` is called and the appropriate shape is created. When lines are selected, the following code in that function is executed:

(File: components/DrawingCanvas.as)

```
if (drawingMode == LINE)
{
    shape = new Line();
    var line:Line = Line(shape);
    line.xFrom = event.localX;
    line.yFrom = event.localY;
    line.xTo = event.localX;
    line.yTo = event.localY;
}
```

This code creates the line object and sets its x and y from/to properties, just like in the previous MXML code example. In this case, we're setting the from/to points to the same point because the user has not yet dragged the mouse, so the start and end points of the line are both set to the location of that initial mouse event.

The line object is then assigned the current `stroke` attribute, which we'll talk about later, and is added to the drawing canvas, which makes it visible in the application window:

```
shape.stroke = stroke;
addElement(shape);
```

A mouse drag caused a call to the `mouseMoveHandler()` function, which then calls `dragTo()`, as we saw earlier. The shape is then changed appropriately. In the case of lines, we simply change the (`xTo`, `yTo`) endpoint:

```
case LINE:
    Line(shape).xTo = dragX;
    Line(shape).yTo = dragY;
    break;
```

When the user releases the mouse button, the `mouseUpHandler()` function is called, which again calls the `dragTo()` function to set a final (`xTo`, `yTo`) endpoint for the line.

That's it for lines. They are very simple objects with just two endpoints. The way that the line between those points is drawn depends on the `stroke` property, which we will discuss in Section 1.

The `Rect` class

Many GUI controls use rectangular graphics. Rectangles are useful for defining the boundaries of objects, like the edges of buttons, or the box of checkboxes, or the borders of panels and windows. Rectangles are also useful for defining the backgrounds of containers, such as a gradient background in an application window. Unlike lines, which have only a `stroke` object to define the line characteristics, rectangles have both outline graphics, defined by a `stroke`, and interior graphics, defined by a `fill`.

The `Rect` object draws a rectangle with optional rounded corners. The dimensions of the shape are determined by its `width` and `height` properties. The dimensions of the rounded corners, if any, are determined by the

radiusX and radiusY properties, which apply to all corners of the rectangle. Some situations call for different rounding on each corner, so there are also overriding properties, such as bottomLeftRadiusX and similar properties for the other corners.

Here is an example of creating a simple black square in MXML:

(File: SimpleObjects.mxml)

```
<s:Rect x="150" y="20" width="80" height="80">
    <s:fill>
        <s:SolidColor color="0"/>
    </s:fill>
</s:Rect>
```

In the Shapely application, rectangles are created as the user drags the cursor when the rectangle drawing mode is selected. In the mouseDownHandler() function, we switch on drawingMode to create the appropriate shape:

(File: components/DrawingCanvas.as)

```
switch (drawingMode)
{
    case RECT:
        shape = new Rect();
        shape.x = event.localX;
        shape.y = event.localY;
        break;
```

Only the location of the object is set; its width and height have the default value of 0, because the user has not yet dragged out the shape to give it dimension. The shape is then filled with the current fill object, which we'll discuss later:

```
FilledElement(shape).fill = fill;
```

Then the object's stroke is set and it is added to the scene, as we saw in the earlier Line section.

As the user drags the mouse around and then releases the mouse button, the dragTo() function is called. This function sets the rectangle's dimensions according to the new mouse position:

```
case RECT:
    shape.width = dragX - shape.x;
    shape.height = dragY - shape.y;
    break;
```

The `Ellipse` class

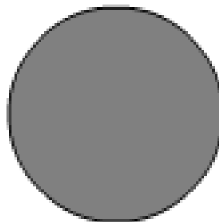
Ellipses are less common in UI controls, although circles (ellipses with equal width and height values) are useful for components like radio buttons. Circular controls are also used in custom UIs, where you may not want your buttons to look like standard rectangular buttons. `Ellipse` is similar to `Rect`; it is positioned with `x` and `y` and sized with `width` and `height`.

This code creates a gray circle with a black outline in MXML:

(File: SimpleObjects.mxml)

```
<s:Ellipse x="20" y="220" width="80" height="80">
    <s:fill>
        <s:SolidColor color="gray"/>
    </s:fill>
    <s:stroke>
        <s:SolidColorStroke color="black"/>
    </s:stroke>
</s:Ellipse>
```

And this is what it looks like:



(Demo: SimpleObjects)

The code for ellipses in `Shapely` is very similar to that for rectangles, because both shapes are positioned and sized in the same way. First, the shape is created based on the value of the `drawingMode` variable in the `mouseDownHandler()` function:

(File: components/DrawingCanvas.as)

```
case ELLIPSE:
    shape = new Ellipse();
    shape.x = event.localX;
    shape.y = event.localY;
    break;
```

On mouse move and up events, the shape is resized according to the new mouse location in the `dragTo()` function:

```
case ELLIPSE:
    shape.width = dragX - shape.x;
    shape.height = dragY - shape.y;
    break;
```

The `Path` class

Lines, rectangles, and ellipses are great for creating simple shapes and lines. But if you want an irregular shape, curved lines, or custom artwork from design tools, you're going to need to draw `Path`s. The `Path` object constructs a path, filled or empty, from a set of line and curve segments. You specify the segment information for a path in the `data` property, which is a String specifying the various move/line/curve pieces that construct the path. An optional `winding` property can be used to specify which side of the path should be filled (if the `fill` property is not `null`).

Several different commands can be used in a `Path`'s `data` string. Each one is abbreviated with a single letter, followed by applicable numerical values.[1] A capital letter indicates the values are absolute coordinates, whereas a lower case letter indicates a position relative to the current position:

move (example: `data="m 10 20"`**)** move the pen to the specified location

line (example: `data="l 10 20"`**)** draw a line to the specified location

horizontal line (example: `data="h 10"`**)** draw a horizontal line to the specified x location (a simplification of the line command)

[1] The syntax and abbreviations used for the string match those of the Path element of SVG (Scalable Vector Graphics), a W3C standard vector API.

vertical line (example: `data="v 20"`**)** draw a vertical line to the specified y location (a simplification of the line command)

quadratic Bézier (example: `data="q 0 0 10 20"`**)** draw a curve[2] with one control point whose x and y are specified first, followed by the x and y location the curve will draw to.

cubic Bézier (example: `data="c 0 0 5 10 10 20"`**)** draw a curve with two control points whose x and y are specified first, followed by the x and y location the curve will draw to.

close path (example: `data="... z"`**)** close off the path by drawing a line to the starting point of the path. This item is optional; if it is not supplied, the path will end at the last data point.

It is important to note that each drawing operation, whether move, line, or curve, starts from the current pen position and ends with the pen in the position specified in the command. So you only need to move the pen explicitly if you wish to start a segment from a different location than its current one. For example, if you want to draw a path that goes from `(0, 0)` over to `(100, 0)` and then down to `(100, 100)`, you could simply type `data="H 100 V 100"`. Here's a simple path constructed in MXML:

(File: SimpleObjects.mxml)

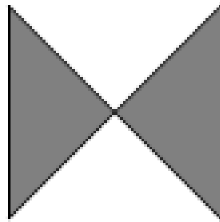
```
<s:Path x="20" y="320" data="L 80 80 V 0 L 0 80 V 0">
    <s:stroke>
        <s:SolidColorStroke color="black"/>
    </s:stroke>
    <s:fill>
        <s:SolidColor color="gray"/>
    </s:fill>
</s:Path>
```

[2] A Bézier curve is specified by *anchor points*, the endpoints of the curve, and *control points*, which specify the path the curve follows between the anchor points. A quadratic Bézier curve starts at one anchor point going in the direction of a single control point and ends at the other anchor point coming from the direction of that control point. A cubic Bézier curve leaves the starting anchor point in a direction of a first control point and arrives at the endpoint in a direction from a second control point. These curves generally do not pass through the control points unless those points lie in a straight line between the anchor points. In the `Path` `data` string, the first anchor point is implicitly specified by the current location of the pen; the curve operations need only specify the control point(s) and ending anchor point.

Path to confusion

It may not seem like hand-coding paths with complex curves would be simple at first glance. But upon using the Path API for a while ... it's still not simple. Instead, paths can be easy to code for simple primitives, but more complex paths will probably come from tools instead. For example, Adobe Illustrator and Adobe Fireworks can export drawings into a format called FXG, which is a simplified form of exactly the graphics primitives detailed in this chapter. Vector paths are used extensively in Illustrator, and these shapes output as Path primitives in the FXG files. So you might want to consider the Path primitive as something that you may encounter in code that is generated by tools, but you probably won't be using it much in your hand-coded graphics.

And this is what that path looks like:



(Demo: SimpleObjects)

Now we'll create Path objects in Shapely. As with the other shapes, paths are created in the mouseDownHandler() function. But we do not yet add data to the path since the user has only told us where it will be located, not where it will draw to next. Instead, we create the pathPoints Vector to hold the points of the Path:

(File: components/DrawingCanvas.as)

```
case PATH:
    shape = new Path();
    pathPoints =
        new <Point>[new Point(event.localX, event.localY)];
    break;
```

When the user moves the mouse or releases the button, the dragTo() function is called, where the new point is added to pathPoints and the function constructPath() is called:

```
case PATH:
    pathPoints[pathPoints.length] = new Point(dragX, dragY);
    constructPath();
    break;
```

The constructPath() function turns the set of points in pathPoints into a data string for the path:

```
private function constructPath():void
{
    var dataString:String = "M " +
        pathPoints[0].x + " " + pathPoints[1].y;
    for (var i:int = 1; i < pathPoints.length; ++i)
    {
        var pt:Point = pathPoints[i];
        dataString += " L " + pt.x + " " + pt.y;
    }
    Path(shape).data = dataString;
}
```

This function walks through pathPoints, turning the first point into a *move* operation and subsequent points into *line* operations in the data string. This process causes a multi-segment line to be created, starting at the point where the user first pressed the mouse (the first point in pathPoints) and continuing through every other point we recorded. Because mouse motion is handled very quickly, this line-segment approach results in a reasonable approximation to freehand scribbling because each straight line segment will only be as long as the distance covered between each mouseMove event.

The reason for the extra layer of indirection with pathPoints is that we cannot simply edit the existing Path shape's points, like we do with the Line, Rect, and Ellipse shapes that we saw earlier. Instead, the only way to change a Path is to supply a new data string. So we record the path's points in a separate data structure and re-create the data string every time we add a new point.

Now that we've seen the different shapes that are possible to create, it's time to talk about the attributes with which the shapes are drawn. Let's learn about `stroke` and `fill`.

Strokes of genius: lines and outlines

You may want to draw lines in your UI to achieve a particular effect, like borders on filled areas, outlines on empty areas, or separator lines between different elements in the interface. These may be straight or curved lines, or bounding lines around larger, filled areas. These lines can be drawn in different ways, with different colors, widths, and joins at the corners. These line properties are defined as *strokes* on the objects.

All of the graphics objects in Flex 4 except for `BitmapImage`, which we'll see later, have an optional `stroke` property that defines the characteristics of the object's lines, like their color and width. For the one-dimensional `Line` object, the `stroke` is all there is. The object's `stroke` is all that you see of the object. For the rest of the objects, `Path`, `Rect`, and `Ellipse`, the `stroke` is the border line around the object's filled area.

Strokes come in three varieties: `SolidColorStroke`, which has a single color, and two strokes that use gradients, `LinearGradientStroke` and `RadialGradientStroke`. You'll see more about gradients later when I discuss `fill` objects. For now, we'll talk about solid color strokes, which is the common case for lines.[3]

A `stroke` has a few properties that are necessary for specific situations, but for which the defaults are generally sufficient. For example, the `joints` and `miterLimit` properties are useful for controlling how the intersections look with multi-segment stroked objects like `Path` and `Rect`. And the `scaleMode` property controls how scaling on the object affects the width of the stroke. Here we'll focus on just the more common stroke properties used to achieve particular effects on the stroke.

`weight` determines the width of the stroke in pixels. A value of 0 is equal to a one-pixel-wide line, but the line stays at that thickness even when scaled. This behavior is in contrast to that when `weight` equals 1, which also results in a one-pixel-wide line on an unscaled object. But

[3] You can see cases of gradients used in strokes in some of the standard Flex 4 component skins like `ButtonSkin`. These skins have very subtle effects that call for rich graphic elements like gradient strokes and fills. But more typical lines and borders are drawn with single colors.

a line with `weight` equal to 1 will scale with the object so an object with a scale factor of 2 will have lines twice as wide as that object with a scale factor of 1.

`color` an unsigned integer value that describes the red, green, and blue (RGB) values that contribute to the final color value. This is a standard RGB representation in an integer, where the bottom-most (least significant) byte represent the blue value, the next byte holds the green value, and the next byte holds the red value. You can picture the color in hex form as the number `0xRRGGBB`. The left-most (most significant) byte of the 32-bit value is unused.

For convenience, the MXML compiler will turn standard color names into the appropriate integer values. You can also use the numeric form of a color in either hex, integer, or HTML-color formats. For example, a value of `"blue"` is equivalent to `"0xff"`, `"255"`, and `"#FF"`.

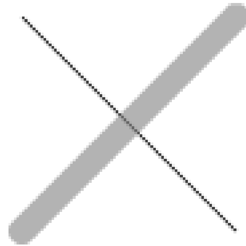
`alpha` the amount of translucency that the object's stroke has. A value of 1 causes the stroke to be completely opaque (nothing behind the object's stroke can be seen through it). A value of 0 causes the stroke to be completely transparent (the stroke is not seen at all, and objects behind it are fully visible). Any value between 0 and 1 causes the stroke to be translucent, allowing both the stroke and the objects behind it to be partially visible, with greater values of alpha making the stroke more opaque. The opacity of the overall object you create is typically controlled with the object's `alpha` property, not the object's stroke's `alpha`, but if you want separate control over the stroke's opacity, use the stroke's `alpha` property.

Here is an example of two lines drawn with different strokes:

```
<s:Line xFrom="20" yFrom="20" xTo="100" yTo="100">
    <s:stroke>
        <s:SolidColorStroke color="black"/>
    </s:stroke>
</s:Line>
<s:Line xFrom="100" yFrom="20" xTo="20" yTo="100">
    <s:stroke>
        <s:SolidColorStroke color="gray"
                alpha=".6" weight="10"/>
```

```
    </s:stroke>
</s:Line>
```

The first object is a black line with the default `weight` (0) and `alpha` (1). The second object is a wide gray that is translucent (note that you can see the black line through the wide gray line), as seen here:

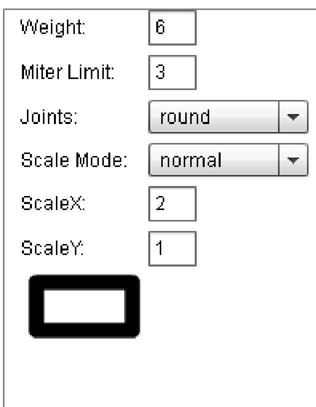


(Demo: SimpleObjects)

The `StrokeTest` application helps visualize how the various stroke parameters affect the look of our stroked primitives. This code draws a `Rect` object with a stroke:

(File: StrokeTest.mxml)
```
<s:Rect x="20" y="170" width="30" height="30"
        scaleX="{Number(scaleXInput.text)}"
        scaleY="{Number(scaleYInput.text)}">
    <s:stroke>
        <s:SolidColorStroke color="black"
                weight="{Number(weightInput.text)}"
                miterLimit="{Number(miterLimitInput.text)}"
                joints="{jointsInput.selectedItem}"
                scaleMode="{scaleModeInput.selectedItem}"/>
    </s:stroke>
</s:Rect>
```

The `Rect` object takes its scale factors from the text input fields so that you can see how scaling in either direction affects the results. The `stroke` object is an instance of `SolidColorStroke` with a color of `black`. The `stroke` object has other properties that are bound to the values of the various controls in the GUI. You can see the results from a nonzero line weight and rounded joints in this screenshot:



(Demo: StrokeTest)

The application is pretty simple as Flex applications go. The interesting part is in how the properties affect the look of the graphic primitive. Be sure to play with it to get a feel for how the properties interact.

Fills: it's what's on the inside that counts

All of the Flex shapes except `Line` can have a `fill` as well as a `stroke`. The `fill` specifies what happens on the interior of the object. So, for example, a rectangle's `stroke` is drawn on the outside of the area and its `fill` is the interior of that area. As with `stroke`, the `fill` property is optional, so any of these objects can have a `stroke` or a `fill` or both or neither (although having neither one makes for a pretty useless shape).

Three types of fills are possible. As with `stroke`, you can fill with a solid color or a gradient. Additionally, you can fill the area with a bitmap image. We'll discuss all of these options next.

Solid color fills

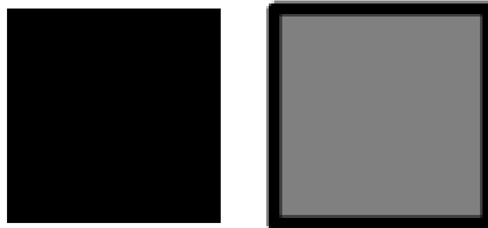
The simplest way to fill an area is with a single solid color. For example, the drawing canvas of `Shapely` is filled with a solid white color. Just like the solid color `stroke` discussed in Section 1, the solid color `fill` has the properties `color` and `alpha`. These properties are exactly the same for both strokes and fills; see Section 1 for more information on them.

Here is an example of two filled rectangles:

(File: SimpleObjects.mxml)

```
<s:Rect x="150" y="20" width="80" height="80">
    <s:fill>
        <s:SolidColor color="0"/>
    </s:fill>
</s:Rect>
<s:Rect x="250" y="20" width="80" height="80">
    <s:fill>
        <s:SolidColor color="black" alpha=".5"/>
    </s:fill>
    <s:stroke>
        <s:SolidColorStroke color="black" weight="5"/>
    </s:stroke>
</s:Rect>
```

The first object is a black-filled rectangle with the default `weight` (0) and `alpha` (1). The second object is filled with translucent black (`alpha=".5"`), making the result gray since the rectangle is drawn over a white background. This second rectangle also has a black, wide `stroke` object. Note that the opacity of the `stroke` object is independent of the `fill`'s opacity.



(Demo: SimpleObjects)

Bitmap fills

Sometimes, you want to fill an area with an image. If you simply want a rectangular image in the scene, it's probably easier to use the `Image` or `BitmapImage` class (which we will see more of later in this chapter). But you can fill any arbitrary shape, like a path, rounded rectangle, or circle, with an image using a `BitmapFill`.

Several properties exist on `BitmapFill` to define the image resource that the fill uses and the way the image is displayed in the filled area:

`source` defines the bitmap that is displayed in the fill. This parameter is flexible and can be used to specify an embedded image file, an instance of a `Bitmap` or `BitmapData` object, or the class name or instance of a display object. Typically, you use an embedded image file, like this: `source="@Embed('tree.jpg')"`.[4]

`smooth` defines whether the image is "smoothed" when it is scaled to a different size than the original bitmap image. By default, `smooth` is false, which results in using the "nearest neighbor" approach, where pixels are chosen from the original image based on which one is closest to the current pixel being drawn. This approach is the fastest option when the image is scaled to fit into the fill area, since it requires no calculations. But scaling without smoothing can result in rendering artifacts. If `smooth` is set to true, scaled images will use a simple bilinear smoothing algorithm, where the pixels to the left, right, top, and bottom of the destination pixel are combined to create a blended pixel value. This property only comes into play when an image is scaled; an image that is displayed in its original resolution will simply use the original pixel values with no smoothing applied.

`fillMode` tells the graphic object how to fill the shape area if the source bitmap is smaller than the shape in either dimension. Three possible values are available, all of which are specified in the `BitmapFillMode` class (or you can choose to use the equivalent strings, like "scale" instead of `BitmapFillMode.SCALE`):

`SCALE` the default value, which causes the bitmap to be scaled (either down or up) to fit the dimensions of the shape that it fills.

[4] The `@Embed` directive tells the compiler to bundle the specified resource with the application (here an image, but `Embed` can be applied to other assets as well). With `BitmapFill`, as well as with the `BitmapImage` object you'll see later in this chapter, any image resource must be embedded. If you use the `Image` control from Flex 3, you can also refer to an image by relative or absolute URL, without embedding the file. If you do not use `Embed`, the image will be loaded when the `Image` component is created, and may not be shown immediately if there is a loading delay. When `Embed` is used, the resource is bundled with the application and is loaded synchronously when the component is created. The `Embed` approach trades off faster image loading time with larger application footprint size, since embedded image assets are packaged into the downloaded application's SWF file.

CLIP causes the bitmap to be drawn in its original size, either being clipped by the size of the region (if the bitmap is larger than the dimensions of the BitmapImage) or leaving empty space (if the bitmap is smaller).

REPEAT causes the bitmap image to repeat or tile itself inside the region, filling the dimensions of the shape.

alpha represents the amount of translucency that the bitmap fill has. This property acts just like the same property on the solid color fill that we discussed earlier.

BitmapFill also has properties for positioning and transforming the bitmap within the filled area. But those parameters are less commonly used and self-explanatory, so I'll defer to the SDK documentation.

Here is a simple example of using a BitmapFill on a rectangle object:

(File: SimpleObjects.mxml)

```
<s:Rect x="350" y="20" width="80" height="80">
    <s:fill>
        <s:BitmapFill
            source="@Embed('images/SanFrancisco.jpg')"/>
    </s:fill>
    <s:stroke>
        <s:SolidColorStroke color="gray" weight="5"/>
    </s:stroke>
</s:Rect>
```

The rectangle has a fill with just one parameter specified: the source. Note that the bitmap, by default, scales to fit the area of the object, so little else is needed unless you want to change the way the image maps into the area.

(Demo: SimpleObjects)

Gradient fills

Gradients are so useful in creating rich UIs that it is worth taking a moment to talk more generally about them before diving into the details of the API of the gradient-based fill classes.

Gradients are used to fill an area with a series of colors. Two types of gradients are supported in Flex: linear and radial. Linear gradients have a color change along one dimension (left to right, top to bottom, or along arbitrary degree of rotation). Radial gradients change colors from some center point out to some perimeter of a circle. Both gradients can be defined with several colors along the way, so that they can change either from one start color to a single end color, or they can change from the start color through a series of other colors (called *gradient entries*, or sometimes, *gradient stops*) along the way before finally reaching the end color.

Gradients provide a simple way to liven up a GUI, from rich backdrops to 3D effects to interesting reflection techniques.

Gradients can be used to liven up a GUI in very simple ways, from providing a rich backdrop to giving components a 3D look, with highlight and shadow effects that really make 2D objects pop out of the screen. Gradients can also be used for some special effects like reflections, where the gradient operates on a translucency value to fade out a reflection for a more realistic look (we'll see this effect in Chapter 3). It's definitely worth learning about the gradient classes so that you can start applying them to your objects and components. And better yet, gradients are much easier to use with the new graphic elements defined in Flex 4, so there's every reason to start using gradients in your rich client applications.

Both types of gradients, linear and radial, use the same method of specifying the set of colors that the gradient transitions between: `GradientEntry`.

The `GradientEntry` class

This class is a simple data structure that holds the information for a particular gradient stop in a linear or radial gradient. For each entry, we need to know the `color`, `alpha`, and `ratio`, which is the point in the overall gradient where the entry's color is sampled at 100%. In other words, the `ratio` is the

point in the overall gradient where the transition from the color in the previous entry to the color in this entry ends and the transition to the next entry's color begins. This information is represented in the following properties:

`alpha` The translucency of the color for this gradient entry. This value acts just like the `alpha` property that we saw earlier for solid color strokes and fills, except that it holds just for this single object in the set of gradient entries instead of for the entire fill.

`color` The color at this point in the gradient, represented as an unsigned integer. This property is just like the `color` property in the solid color stroke and fill discussed earlier, except that this color is true just for this entry and not for the whole fill.

`ratio` The point in the gradient where this entry is applied. This is a percentage value, with 0 representing the start of the gradient and 1 representing the end of the gradient.

A gradient (either linear or radial) consists of a set of `GradientEntry` objects which define how the color and translucency of the gradient changes over the course of the object it fills.

For example, this set of entries defines a gradient that changes smoothly from black to white to gray:

```
<s:GradientEntry color="black"/>
<s:GradientEntry color="white"/>
<s:GradientEntry color="gray"/>
```

Note that this code does not set a `ratio` value for any of the entries. By default, the entries spread themselves equally over the available area. Not defining ratios for these three entries is equivalent to specifying a ratio of 0 for the black entry, .5 for the white entry, and 1 for the final gray entry.

Linear and radial gradient shared properties

Most of the functionality of linear and radial gradients is shared in the common superclass, `GradientBase`. These are the more commonly used shared properties of that class:

`entries` This property defines an `Array` of `GradientEntry` objects, as we saw in the previous section.

`rotation` This property defines the angle of rotation, in degrees, along which the gradient proceeds. By default, gradients move from left to right, horizontally. For example, the black/white/gray gradient entries example in the previous section would, by default, show up with black at the left, white in the middle, and gray at the right. A gradient moving in a different direction is defined using the `rotation` property. For example, a vertical gradient is defined by setting `rotation` to 90. Vertical gradients are more common in UI elements because gradients are often used to give a pseudo 3D lighting effect, where the virtual light source is somewhere above the scene.

`spreadMethod` This property defines what happens outside of the defined gradient area. If the area covered by a gradient does not completely cover its target object, then it needs to know how to color the remaining pixels in the object's area. This property has three possible values, from the `SpreadMethod` class: `CAP`, `REPEAT`, and `REFLECT`. `CAP` causes the color values at the end of the gradients to extend to the boundaries of the filled area. `REPEAT` causes the gradient to repeat itself over and over to fill the target area. `REFLECT` is like `REPEAT`, except each time it repeats it reverses itself.

There are also properties for positioning the starting point of the gradient within the filled area (`x` and `y`), a property for changing the method of color interpolation (`interpolationMethod`), and a property for performing more complex transformations of the gradient fill (`matrix`). I'll just refer you to the Flex SDK documentation for these less commonly-used properties.

That's it for the shared properties. Now let's see how all of this gets put together in the linear and radial gradient objects, along with some examples of the visual results.

The `LinearGradient` class

Linear gradient fills transition through their colors along a straight line. This type of gradient is useful for backgrounds that are much richer than solid colors. Linear gradients are also useful for some 3D effects, such as making UI components look convex or concave, because they are good at mimicking shadows and highlight drop-off.

The `LinearGradient` class provides a single property in addition to those inherited from `GradientBase`: `scaleX`. This property is responsible

for defining the scale factor of the gradient, which is an easy way to define the area covered by the gradient. By default, the gradient fills the area of the target object, but this scale factor can be used to define the gradient pattern over a larger or smaller area. Note that the scale factor is only in the x direction; no scaling in the y direction exists since the gradient only operates in one dimension. So if you want the gradient to be half the size of a 100-pixel wide shape that it fills, you set `scaleX` = 50.[5]

Here are some simple examples that show different linear gradient fills inside `Rect` objects:

(File: SimpleObjects.mxml)

```
<s:Rect x="20" y="120" width="80" height="80">
    <s:fill>
        <s:LinearGradient>
            <s:GradientEntry color="black"/>
            <s:GradientEntry color="white"/>
            <s:GradientEntry color="gray"/>
        </s:LinearGradient>
    </s:fill>
</s:Rect>
<s:Rect x="120" y="120" width="80" height="80">
    <s:fill>
        <s:LinearGradient rotation="90">
            <s:GradientEntry color="0xb0b0b0"/>
            <s:GradientEntry color="0x404040"/>
        </s:LinearGradient>
```

[5] This use of `scaleX` seemed non-intuitive to me when I first saw it. I'm used to the `scaleX` and `scaleY` properties, which are on Flash display objects and Flex components, representing a proportion of an object's pixel size. So if an object has a width of 100 and I want it to be 50 pixels wide on the screen, I expect to set a `scaleX` value of .5. But with gradient fills, it doesn't work that way. If a gradient fills an area 100 pixels wide, but I want it to stop at 50 pixels, I set `scaleX` to 50. What's up with that?

It turns out that `scaleX` for gradient fills means exactly the same thing that it does for display objects; it is a proportion of that object's current size. But the key to understanding `scaleX` with gradient fills is that a gradient fill has a natural size of *one* pixel. So by specifying a `scaleX` value of 50, we're actually saying that the gradient should fill 50 times its natural size, or 50 pixels. One of the confusing things here is that if you don't specify any value for `scaleX`, it fills its object completely. But this is not because the gradient fill has a scale value of 1 (as do typical objects in Flex and Flash). Instead, the `scaleX` property has a default value of `NaN`, which tells the gradient to fill whatever area it occupies, regardless of size.

```
        </s:fill>
    </s:Rect>
    <s:Rect x="220" y="120" width="80" height="80">
        <s:fill>
            <s:LinearGradient rotation="90">
                <s:GradientEntry color="0x808080"/>
                <s:GradientEntry color="0xa0a0a0" ratio=".25"/>
                <s:GradientEntry color="0x202020"/>
            </s:LinearGradient>
        </s:fill>
    </s:Rect>
    <s:Rect x="320" y="120" width="80" height="80">
        <s:fill>
            <s:LinearGradient rotation="90">
                <s:GradientEntry color="0x808080"/>
                <s:GradientEntry color="0x202020" ratio=".1"/>
                <s:GradientEntry color="0x404040" ratio=".75"/>
                <s:GradientEntry color="0xa0a0a0"/>
            </s:LinearGradient>
        </s:fill>
    </s:Rect>
```

This code results in the following:

(Demo: SimpleObjects)

The first of these rectangles is the result from the same black/white/gray gradient entries that we saw earlier. This example uses the default rotation, so the linear gradient proceeds from left to right. The second example shows a subtle vertical gradient between two shades of gray, caused by using a `rotation` value of 90. This gradient is appropriate for some application window and container backgrounds.

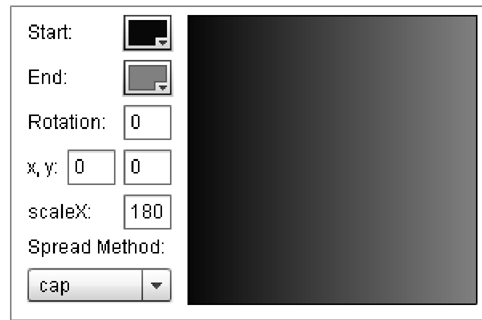
The third and fourth examples show the pseudo-3D effects that linear gradients are sometimes used for. The third object simulates a convex object lit from above, where the light shows most at the top of the object. The rounded effect is achieved by having the gradient proceed from one color to a lighter color at a `ratio` of .25, then down to a darker color at the bottom. The final object shows more of a concave effect, with the light showing most at the bottom of the object.

To see a slightly more involved example, take a look at the example `LinearGradientProperties`. The application uses several GUI controls to allow the user to change the gradient colors, the `rotation`, and other properties of the gradient. The gradient is specified with data bindings to those input values, such as the gradient's `rotation` property being set by the `rotationInput` text control, and fills a `Rect` object as follows:

(File: LinearGradientProperties.mxml)

```
<s:Rect id="rect" width="180" height="180">
    <s:stroke>
        <s:SolidColorStroke color="black"/>
    </s:stroke>
    <s:fill>
        <s:LinearGradient
           rotation="{Number(rotationInput.text)}"
           x="{Number(xInput.text)}"
           y="{Number(yInput.text)}"
           scaleX="{Number(scaleXInput.text)}"
           spreadMethod="{spreadMethodInput.selectedItem}">
           <s:GradientEntry
                 color="{startColor.selectedColor}"/>
           <s:GradientEntry
                 color="{endColor.selectedColor}"/>
        </s:LinearGradient>
    </s:fill>
</s:Rect>
```

The `stroke` on the `Rect` is defined just to give the shape a visual boundary. When you run the application, you can play with various properties of the gradient to see how they affect the visual result, as seen here:



(Demo: LinearGradientProperties)

The `RadialGradient` class

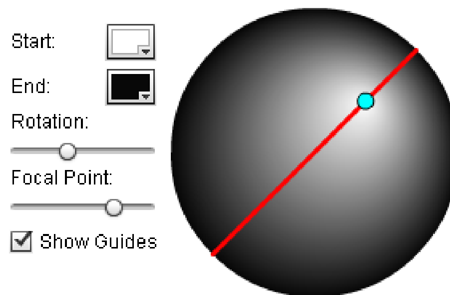
Radial gradients are useful for some special effects like specular highlights and spotlights. They help give graphical objects a 3D look by mimicking circular shadows and highlight drop-off. They are also good for emphasizing areas of focus through spotlight effects.

Radial gradients in Flex are handled with the `RadialGradient` class, which sets up a gradient to start at some center point and radiate outwards to end at the perimeter of the filled area. This class has three properties, beyond the shared ones in `GradientBase`, that help define the way that the gradient fills the area. The `scaleX` and `scaleY` properties act like the `scaleX` property in `LinearGradient`, but since this is a two-dimensional fill, scales happen in two directions. Like the linear gradient's `scaleX` property, these properties default to a value of `NaN`, which causes the gradient to fill the entire area of the object that it is assigned to. So if you don't need to change that behavior, you won't need to set these properties.

The other property of `RadialGradient` is `focalPointRatio`, which is used in conjunction with the `rotation` property to set the location of the center point from which the gradient radiates. The gradient radiates outward toward the boundaries of the gradient area, starting from some point inside. That point is determined by the `rotation` parameter, which tells the gradient the degrees to rotate, and the `focalPointRatio`, which tells the gradient where on that rotation axis to place the center. The `focalPointRatio` is a value from -1 to 1, with -1 placing the point on the left edge of the gradient area and 1 placing it on the right edge. A value of 0, the default for this property, places the value in the middle of the gradient area. Meanwhile, the `rotation` property determines the angle of the center line, with the default

value of 0 being no rotation, so the center line simply extends from left to right through the middle of the gradient area. For example, a rotation of 45 and a focalPointRatio of .5 will place the center of the gradient at the lower right corner of the gradient area.

You can play with the relationship of rotation and focalPointRatio in the RadialGradientProperties demo. Besides showing how these properties affect the look of the gradient, the application has optional guides to display the current rotation (the line through the middle of the circle) and focalPointRatio (the small circle on top of the line). For example, this screen shot shows a gradient with a rotation of 45 degrees and a focalPointRatio of .5:



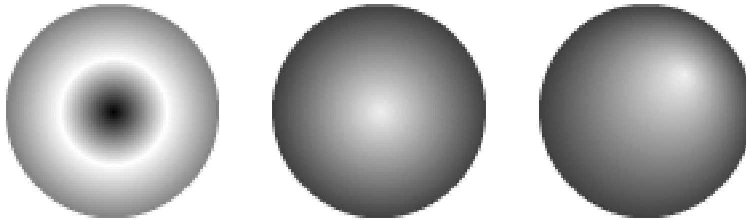
(Demo: RadialGradientProperties)

You can see some simple examples of radial gradients in the following code from SimpleObjects:

(File: SimpleObjects.mxml)
```
<s:Ellipse x="120" y="220" width="80" height="80">
    <s:fill>
        <s:RadialGradient>
            <s:GradientEntry color="black"/>
            <s:GradientEntry color="white"/>
            <s:GradientEntry color="gray"/>
        </s:RadialGradient>
    </s:fill>
</s:Ellipse>
<s:Ellipse x="220" y="220" width="80" height="80">
    <s:fill>
        <s:RadialGradient>
```

```
                <s:GradientEntry color="0xf0f0f0"/>
                <s:GradientEntry color="0x404040"/>
            </s:RadialGradient>
        </s:fill>
    </s:Ellipse>
    <s:Ellipse x="320" y="220" width="80" height="80">
        <s:fill>
            <s:RadialGradient rotation="-45"
                    focalPointRatio=".5">
                <s:GradientEntry color="0xf0f0f0"/>
                <s:GradientEntry color="0x404040"/>
            </s:RadialGradient>
        </s:fill>
    </s:Ellipse>
```

This code results in the following:



(Demo: SimpleObjects)

The first circle uses the gradient entries used in previous examples, where the gradient starts at black, in the center of the circle, transitions through white halfway through, and ends at gray at the perimeter of the circle. It's not a very effective use of this gradient; I just used it for comparison purposes to the earlier examples. The other two circles are more representative of the power of radial gradients.

The second circle transitions from a light gray color in the center to a darker gray at the edge. This simple, two-color gradient effect really pops the circle out of the page, giving it a 3D look that belies its simple composition. The reason that it works so well is that that lighter color in the middle acts just like a *specular highlight*. A specular highlight is the reflection of a light source on an object. On a 3D object with a matte surface, the specular highlight of most light sources, like the sun, show up as bright spots that

gradually fade toward their edges to the normal object color. The radial gradient mimics a specular highlight because it is lighter in the center and falls off smoothly toward a darker color at the edges.

The third example takes the circle a step closer toward mimicking reality. The second example works, but only if you don't actually think about the light source. If you stop and think about it, it doesn't make much sense; the light seems to be coming from the viewer. Unless the viewer is wearing a miner's helmet with a light shining directly out from their forehead,[6] it's not very realistic.

A typical light source is usually one that shines from above, like the sun or the lights in a room. And a typical light source also isn't usually so symmetrically located on the vertical plane between the viewer and the object being lit. The third circle addresses these problems by offsetting the gradient center, and therefore the virtual light source, to the upper-right of the object. It's a subtle change from the second example, but I like it because it gives a more real-world feel to the object.

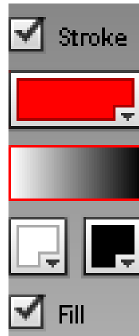
Radial gradients that are offset from dead center of the object look more natural; the real world rarely lights objects from the direction of the viewer.

Setting strokes and fills in `Shapely`

Now that we've talked about stroking and filling objects, we're finally able to discuss how the `Shapely` application sets the drawing state that is used when creating graphics shapes.

The components at the bottom of `Shapely`'s control panel determine the stroke and fill attributes that are used when drawing. The checkbox at the top controls whether a stroke is used and the checkbox at the bottom controls whether a fill is used. In between these components are `ColorPickers` for the stroke and the fill gradient. And a sample rectangle between the stroke and fill sections shows the user a preview of what shapes look like with the current stroke and fill settings:

[6] This is probably not a demographic that is worth targeting in general, although such users could be interesting for data-mining applications.



(Demo: Shapely)

These objects are created by the following code:

(File: components/ControlPanel.mxml)
```
<s:CheckBox fontSize="9" label="Stroke" id="strokeCB"
        selected="true" change="setDrawingState()"/>
<mx:ColorPicker id="strokeColor" change="setDrawingState()"
                width="100%" selectedColor="0xff0000"/>
<s:Rect id="sampleRect" x="10" width="100%" height="20"/>
<s:HGroup enabled="{fillCB.selected}">
    <mx:ColorPicker id="fillColor"
                change="setDrawingState()"
                selectedColor="0xffffff"/>
    <mx:ColorPicker id="fillGradientColor"
                change="setDrawingState()"
                selectedColor="0x0"/>
</s:HGroup>
<s:CheckBox id="fillCB" label="Fill" fontSize="9"
                change="setDrawingState()"/>
```

When any of these stroke and fill settings change, the `setDrawingState()` event handler function is called:

```
private function setDrawingState():void
{
    var newStroke:IStroke;
    var newFill:IFill;
    if (fillCB.selected)
    {
```

```
            if (fillColor.selectedColor ==
                    fillGradientColor.selectedColor)
                newFill = new SolidColor(
                     fillColor.selectedColor);
            else
            {
                newFill = new LinearGradient();
                LinearGradient(newFill).entries = [
                    new GradientEntry(
                        fillColor.selectedColor),
                    new GradientEntry(
                        fillGradientColor.selectedColor)
                ];
            }
        }
        if (strokeCB.selected)
            newStroke = new SolidColorStroke(
                strokeColor.selectedColor);
        sampleRect.stroke = newStroke;
        sampleRect.fill = newFill;
        var drawingChangeEvent:DrawingStateChangeEvent =
            new DrawingStateChangeEvent("drawingStateChange",
            newStroke, newFill);
        dispatchEvent(drawingChangeEvent);
    }
```

The setDrawingState() function sets up the new stroke and fill objects to be used by both the sampleRect visible in the control panel and future shapes that are drawn to the canvas. If the stroke checkbox strokeCB is not selected, the newStroke object will be null and both sampleRect and future shapes will not be drawn with a stroke. The same thing is true for fills and the newFill object, based on whether fillCB is selected.

If a stroke is selected, it is set to a simple SolidColorStroke based on the color selected in the strokeColor ColorPicker control. Fills are a bit more complicated. To simplify the UI and the explanation of how Shapely works, the fill color is always specified in terms of a gradient, with a separate ColorPicker for each color. If both gradient colors are the same, then a SolidColor fill is created with that color. Otherwise, a LinearGradient

fill is created. Note that this gradient is always left-to-right; no option exists to change the gradient direction. Also, the user cannot choose more than two entries in the gradient and the gradient is always a `LinearGradient`, never a `RadialGradient`. These were conscious decisions made to limit the complexity of the UI and the code. Changing any or all of these options is hereby left as an exercise for the reader.[7] It shouldn't be difficult to add these features using what we learned in this chapter.

Once we've set values for the `newStroke` and `newFill` objects, we create a `DrawingStateChangeEvent`, which is a simple `Event` subclass that contains the new fill and stroke objects to be sent to the event's listeners. We dispatch this event, which is received by the `drawingStateChange()` function in `Shapely`:

(File: Shapely.mxml)

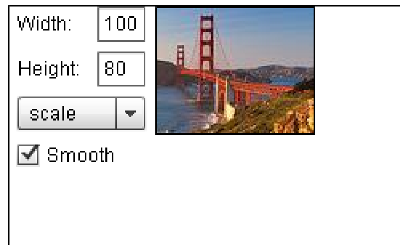
```
private function drawingStateChange(
        event:DrawingStateChangeEvent):void
{
    canvas.stroke = event.stroke;
    canvas.fill = event.fill;
}
```

Image is everything

One graphic area that we haven't covered yet, but which is no less important than the vector-based shapes we discussed earlier, is images. Images can be useful in many different places in rich client applications, from the icons in buttons to photographs in media applications. Images can also be useful in ways that aren't obvious, such as capturing GUI objects as bitmap images and manipulating those objects in visually interesting ways (a technique that we will see applied later when we discuss Pixel Bender shader-based animation effects in Chapter 10).

[7] I've always wanted to say "left as an exercise for the reader." Too many years of math classes with infuriatingly non-obvious proofs in the textbooks marked with that catch phrase engendered a sense of vengeance which is only overcome through propagating the same phrase through my books. But hopefully my use is a bit less devious; I do think that the details here are obvious and doable. It's just that they just require more work and code than is worth delving into in the pages of this book, especially for the goal we're trying to achieve here, which is knowledge of how the graphics classes work.

Here's a simple application that displays an image, along with controls that let the user change the way the image is rendered:



(Demo: BitmapImageTest)

The image control in the application is a `BitmapImage` object, with its properties determined by the values in the UI controls:

(File: BitmapImageTest.mxml)
```
<s:BitmapImage source="@Embed(source='images/Bridge.jpg')"
       smooth="{smoothInput.selected}"
       fillMode="{fillModeInput.selectedItem}"
       width="{Number(widthInput.text)}"
       height="{Number(heightInput.text)}"/>
```

The `Image` and `BitmapImage` controls display images in a GUI. We focus on `BitmapImage` in this chapter, but you may also want to look at the `Image` class for your applications. An important limitation exists for `BitmapImage`; it can only handle embedded assets (where the bitmap supplied to the object is loaded at compile time and stored as an asset with the application). If you want to dynamically load image assets (such as from a network location), then you'll want to look into using `Image` instead. Most of the demos in this book use embedded assets, so the simpler `BitmapImage` class does the trick.

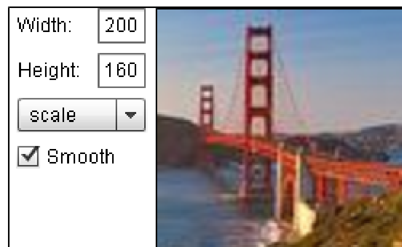
A `BitmapImage` object displays a specified bitmap in a given position (`x` and `y`) and size (`width` and `height`). `BitmapImage` also has the same three properties `source`, `smooth`, and `fillMode` that are on `BitmapFill`, so you might want to refer to the section on `BitmapFill` earlier in this chapter for information on these properties.

We can see the results of the different fill modes in the demo application, `BitmapImageTest`. When the user selects different `width` and `height` values, the size of the `BitmapImage` object changes to the new dimensions.

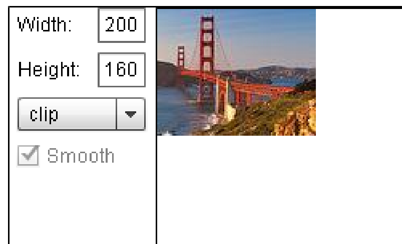
To make it more obvious what's going on in the window, a bounding rectangle is drawn at the selected size. Setting the `left`, `top`, `right`, and `bottom` values all to 0 pins the rectangle to the boundaries of its containing group, which is sized according to the dimensions of the `BitmapImage` object, so the rectangle assumes that same size:

```
<s:Rect left="0" top="0" right="0" bottom="0">
    <s:stroke>
        <s:SolidColorStroke color="black"/>
    </s:stroke>
</s:Rect>
```

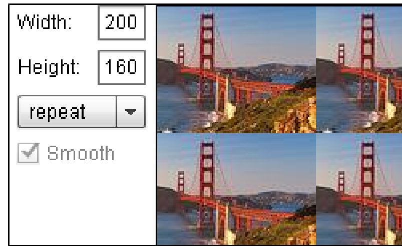
When the size is doubled, the image scales to fill the new size:



When the `fillMode` is changed to `clip`, the bitmap stays at its original size, even though the space it occupies (which we can see from the border rectangle) is much larger:



When `repeat` is chosen as the `fillMode`, the bitmap is repeated across the size of the `BitmapImage` space:



You can also play around with the `Smooth` checkbox to see the pixelization artifacts that result from not smoothing during scaling operations. The impact of these artifacts varies based on the original image, the size of that image, and the scaling factor.

Reflections, like gradient fills, are one way to make a 2D interface more rich and 3D-like, by giving the user the impression that the objects interact like they would in the real world.

As one final view of how you might use bitmaps and graphics in different ways in an application (and as a subtle teaser for techniques that we will elaborate on in the next chapter), let's see how to create a simple reflection effect. First, we need a rich background for our application which we'll get with a gradient fill:

(File: Reflexion.mxml)

```
<s:Rect width="100%" height="100%">
    <s:fill>
        <s:LinearGradient rotation="90">
            <s:GradientEntry color="0x404040"/>
            <s:GradientEntry color="0xf0f0f0"/>
        </s:LinearGradient>
    </s:fill>
</s:Rect>
```

Next, we want to display an image with a reflection of itself. This is done with a `VGroup`, which automatically stacks the two objects (the image and its reflection) vertically. The reflection is exactly the same image, but reflected vertically, so it is scaled it in the y direction:

```
<s:VGroup id="reflectionContainer" x="50" y="50" gap="0">
    <s:BitmapImage id="image" source="{Harbor}"
           x="50" y="50" width="400" height="200"/>
    <s:BitmapImage source="{Harbor}" scaleY="-1" alpha=".4"
           width="{image.width}" height="{image.height}"/>
</s:VGroup>
```

You should note a couple of things about this code. First, the `gap` in the `VGroup` is set to 0 because the reflection should start exactly where the image stops (unless we are trying to mimic the object floating above the reflected surface). Second, the way that the reflection is achieved is by scaling the image by -1 in y. This scaling operation effectively inverts the image vertically, which is exactly what we want. Third, a fractional `alpha` value is set on the reflection to make it translucent. This is necessary because true reflections are never perfect, unless the reflecting surface is a mirror. We want to mimic an imperfect reflecting surface, so we dim the reflected image by giving its `alpha` property a translucent value. The effect is easy to achieve, and provides a reasonable, if simple, approximation of a reflected image:

(Demo: Reflexion)

Reflections in the real world are never perfect; the more we can mimic real-world reflection effects, the more natural they will seem to the user.

We can improve on this effect, however. The translucent reflection in this effect, while better than a fully opaque version, just isn't real enough. There are other things that we can do to make the reflection much more realistic. But these techniques require knowledge of Flex filters, which is both an interesting topic and excellent segue to the next chapter.

Conclusion

In this chapter, we saw how Flex 4 allows you to create shapes with different stroke and fill properties to create custom graphics for you application, in either MXML or ActionScript code. You can use these drawing primitives to create anything from drawing applications to image viewers. These graphics shapes and attributes are also useful for creating custom component skins, as we will see in Chapter 6.

In the next chapter, we will see how to use Flex filters to add rich graphical effects to your applications.