Foundation ActionScript 3.0 Image Effects

Todd Yard

Foundation ActionScript 3.0 Image Effects

ISBN-13 (pbk): 978-1-4302-1871-5

ISBN-13 (electronic): 978-1-4302-1872-2

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com.

For information on translations, please contact Apress directly at 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at http://www.apress.com/info/bulksales.

The source code for this book is freely available to readers at www.friendsofed.com in the Downloads section.

Credits

Lead Editor
Ben Renow-Clarke

Technical Reviewers
Brian Deitte, Chris Pelsor

Editorial Board
Clay Andres, Steve Anglin, Mark Beckner, Ewan Buckingham, Tony Campbell, Gary Cornell, Jonathan Gennick, Michelle Lowman, Matthew Moodie, Jeffrey Pepper, Frank Pohlmann, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh

Project Manager
Beth Christmas

Copy Editor
Heather Lang

Associate Production Director
Kari Brooks-Copony

Production Editor
Kelly Winquist

Compositor
Molly Sharp

Proofreader
April Eddy

Indexer
Carol Burbo

Artist
April Milne

Cover Image Designer
Corné van Dooren

Interior and Cover Designer
Kurt Krames

Manufacturing Director
Tom Debolski

For every small-town, wide-eyed pixel that ever stepped off a train in the big city with nothing but a suitcase of colors and a dream to light up the screen.

CONTENTS AT A GLANCE

CONTENTS

ABOUT THE AUTHOR

Todd Yard is software architect at Brightcove in Cambridge, Massachusetts, where he has worked since its early days in 2005 when everyone could fit into a small room. There, he is focused on the front-end rich media player framework for Brightcove's media management and distribution service. Prior to that, he was a partner with ego7 in New York and lead developer for its Flash content management system and community application suite. Sometime in the midst of all that, he developed applications, animations, and advertisements for a range of clients including GE, IBM, AT&T, and Mars.

As an author, Todd has previously contributed to 13 Flash and ActionScript books from friends of ED, including *Object-Oriented ActionScript 3.0* and *Extending Flash MX 2004*, and has served as technical editor on four others. He has also contributed a number of articles on Flash, Photoshop, and Illustrator to the *WebDesigner* and *Practical Web Projects* magazines.

When Todd is not at a computer, he might be found on a local stage somewhere singing and, on occasion, dancing. That's what he used to do before he was lured by the siren call of software development.

His personal web site can be found at `http://www.27Bobs.com`.

ABOUT THE TECHNICAL REVIEWERS

Brian Deitte is a software developer working at Brightcove, where he has helped to create a video mash-up tool built in Flex called Aftermix and various advertising solutions. Previously, he worked at Adobe on the Flex SDK team, from Flex 1.0 to Flex 2.01. He keeps a blog at `http://deitte.com`.

Chris Pelsor is an award-winning developer and manager of Tarantell's experiential technologies group, Tarantell:Hybrid. He has helped develop solutions for Adobe, Microsoft, Sony, and Jameson Whiskey. When he isn't busy sitting on a train for eight hours a week, he spends his free time perfecting his pale ale recipe and parenting two children with his partner in crime Lisa in Heidal, Norway, also known as the middle of nowhere.

ABOUT THE COVER IMAGE DESIGNER

Corné van Dooren designed the front cover image for this book. After taking a brief from friends of ED to create a new design for the Foundation series, he worked at combining technological and organic forms, with the results now appearing on this and other books' covers.

Corné spent his childhood drawing on everything at hand and then began exploring the infinite world of multimedia—and his journey of discovery hasn't stopped since. His mantra has always been "The only limit to multimedia is the imagination," a saying that keeps him moving forward constantly.

Corné works for many international clients, writes features for multimedia magazines, reviews and tests software, authors multimedia studies, and works on many other friends of ED books. You can see more of his work at and contact him through his web site, `www.cornevandooren.com`.

If you like Corné's work, be sure to check out his chapter in *New Masters of Photoshop: Volume 2* (friends of ED, 2004).

INTRODUCTION

I don't think they could have come up with a better name than "Flash." Although that name is now applicable to an entire platform that includes much more than the Flash Player and IDE, at the end of the day, whether you are developing in Flash or Flex Builder, and whether you are delivering online or through AIR, the end result is still a SWF that is rendered in the Flash Player, just as it was when Flash simply made animations. So whatever you produce—movie or application or generative art—is still "Flash." And boy, can it ever. From the very beginning, Flash movies have often evoked reactions of "Wow" and, more apropos to developers like you, "How'd they do that?"

I began working with Flash in 2000, right before the release of Flash 5. If you're an old-timer, you may recall that Flash 4 was still basically a timeline animation tool with only a handful of ActionScript commands available (I liken working in ActionScript without objects and arrays to working in Photoshop before there were layers). And yet the amount of creative and stunning Flash work that was produced in that time with those few commands was truly exciting.

That excitement, and the fact that it could be produced with little programming knowledge, attracted an amazing diversity of talent to Flash, from animators to programmers to graphic artists to musicians. What software today could do the same? This mixture of disciplines helped to create a community that explored this new technology from so many different points of view, a community that openly and happily shared its collective findings. And the end result of all this work and play and experimentation was—and still is—something to look at.

It might be an animation, a game, an application, or just something cool to ogle, but Flash produces visuals. That's what excited me when I first began nearly a decade ago and what has hooked so many others as well. The Flash platform has grown tremendously, and ActionScript, now in its third iteration, is a complex and powerful language that allows developers to create web applications to rival those built for the desktop. And with that power comes greater and finer control over the graphic elements in a Flash movie.

In this book, I will explore the myriad ways that ActionScript allows you to create and control these visuals in a SWF. Whether you produce games, applications, or cartoons, or you just want to play, this book will give you what you need to know to manipulate the pixels to your advantage.

In the first part of this book, we will step through each of the major areas of image creation and manipulation that ActionScript makes available to developers. We'll start at the drawing API, explore filters and blend modes, come to grips with BitmapData and all it provides, and then dive in depth into new features of Flash Player 10 with 3D and Pixel Bender. Once these

topics are covered, we'll look at an effects library that you can use to easily create a multitude of effects through the remaining chapters of this book and beyond.

The second part of this book presents a collection of tutorials that will allow us to explore how to apply the knowledge gained in the first part. That's where we'll start to have some real fun. That's important, because it's the fun in creating with Flash that first drew so many in, myself included, and it's the fun that drives people to play, explore, and come up with some truly wonderful and creative applications. You can still build utilitarian pieces of software that have that Flash—not for general eye candy but to enhance usability, create more immersive and responsive experiences, and generally produce work that goes beyond everything else presented on the Web. This book shows you how.

Intended audience

This is not a book on object-oriented programming or ActionScript 3.0 fundamentals, nor is it a book on the Flash or Flex Builder authoring environments (in fact, in this book, I strive to have very little reliance at all on any specific IDE). You should come with familiarity of how to compile a SWF and have at least intermediate knowledge of ActionScript 3.0.

With this book, I hope to explore the fun to be had when programming graphics and share the enjoyment that I find when pushing the pixel. I was not a computer science major, nor do I come from a programming or mathematics background—I was just enamored with what Flash could do, and it pulled me in completely and hasn't let go. If you are familiar with ActionScript 3.0, are interested in all the graphic capabilities of the language, and aren't scared of some math here and there, then this book is for you.

Development environment

I'm an ActionScript developer. I go from using Flash to Flex Builder to the command-line mxmlc compiler in order to compile my SWFs nearly every day. In this book, I try to present pure ActionScript examples that can be compiled in any of the environments that do not rely on the timeline or library in Flash or on the Flex framework.

As you go through this book, you will find that examples are presented in pure ActionScript 3.0, usually with instructions to compile the SWF or to test the movie without further instructions on how to do so, since this differs in each environment. In this book's appendix, I discuss how you would work in both Flash and Flex Builder when using this book. You should only have to look at it once if you are not already familiar with how to compile with your method of choice.

If you are using Flex Builder 3, you will need to go through a few additional steps to set up the SDK that includes the new ActionScript classes available for Flash Player 10 and configure Flex Builder to compile for this version of the player. If you have not already been using these new classes, like Vector and Shader, you should have a look at the appendix now to see how to set this all up.

For the first couple of chapters, I will point to the appendix as a reminder, but if you jump through this book nonlinearly, you might take a peek at this appendix first.

Code comments

Code should contain helpful comments. I do not believe there is dispute over that. However, you may notice that the code presented within this book does not contain comments, at least in the chapter text itself. There are three reasons for this. First, it saves a heck of a lot of space not having comments, and including comments within these pages would have meant having to drop whole tutorials and present fewer examples. I wanted to present to you as much as was possible for the page limit. Second, and this is personal preference, when I am reading a book exploring new techniques, I often find a lot of code comments in the book text to be overly distracting from the code I am attempting to absorb, as the comments can double the size of code listings and make the actual code more difficult to focus on. Finally, in this book I spend a good amount of text before and after code listings, but outside of the actual code itself, detailing everything that is going on within the code. Including comments would mean either cutting these larger explanations or presenting redundant information.

But, as I said, code should contain helpful comments. You will find, in this book's support files, that all of the classes have been fully commented if you are looking at the code itself. It is only in chapters themselves that the code is presented without the comments.

Layout conventions

To keep this book as clear and easy to follow as possible, the following text conventions are used throughout:

Code is presented in `fixed-width font`.

New or changed code is normally presented in **`bold fixed-width font`**.

Menu commands are written in the form Menu ➤ Submenu ➤ Submenu.

Where I want to draw your attention to something, I've highlighted it like this:

> *Ahem, don't say I didn't warn you.*

Sometimes code won't fit on a single line in a book. Where this happens, I use an arrow like this: ➥.

```
This is a very, very long section of code that should be written all ➥
on the same line without a break.
```

Chapter 1

THE DRAWING API

Back before the earth cooled and life came crawling up from the seas, when the continents of the world were joined in a massive landmass and iPods held less than a gigabyte, Flash offered no way to dynamically create graphics. This would be a very short book if that had remained the case. Thankfully, Flash MX came out and blessed developers with its implementation of a drawing API (application programming interface), and we looked on it and saw that it was good.

A brief history of the drawing API

The original ActionScript 1.0 drawing API allowed for runtime creation of graphics using a small collection of eight simple commands to draw lines and fills. Drawing straight lines and curves and filling these with solid colors or gradients might seem a small thing in today's age of flying cars and jetpacks, but when the previous option

was only to use predrawn vectors and bitmaps from the Library, a drawing API was a boon that offered countless new possibilities, from liquid interfaces to graphs and charting to complex 3D engines to dynamic visualizations.

With the introduction of ActionScript 2.0, the drawing API was given two new commands, one for drawing gradient strokes and the other for filling shapes with bitmaps. A few of the older methods were beefed up as well to offer additional functionality like defining gradient spread methods and line join and cap styles.

With ActionScript 3.0 and the new player runtime that allowed for it in Flash Player 9, all of the drawing methods were moved into their own Graphics class, as opposed to being a part of MovieClip. New methods for drawing some basic shapes, namely rectangles and ellipses, were also added. These alterations were useful but not world changing. Developers waited with bated breath.

Now, with Flash Player 10, ActionScript 3.0's drawing API gets an even more significant overhaul. Drawn graphics from one object can be easily copied into another. Whole sequences of drawing commands can be saved and rerun, even in multiple objects, at any time. Strokes, like fills, can now be filled with bitmaps. Finally, ActionScript's new shaders can be used for both strokes and fills, allowing for custom bitmap gradients and patterns.

The Graphics class and all it offers is a huge subject for a single chapter and an extremely important one. Much of the graphic manipulation we will do throughout this book relies on a firm understanding of what the methods of ActionScript's drawing API provide. So let's start at the very beginning—which is, I hear, a very good place to start—with the original eight simple commands.

What's come before

First, we will look at the methods that have been present in the drawing API from previous versions of Flash. If you are an ActionScript veteran and are familiar with the drawing API from ActionScript 2.0, feel free to skip over this section and proceed right to the new capabilities. I promise not to get offended.

If you have never used the drawing API before or would like a refresher, this section will be a quick run-through of what previous versions of ActionScript and Flash have offered. This won't be an extensive, exhaustive tutorial, since a lot of what it covers has been around for a good long while now, and I want to reserve the beef of the chapter for all the lovely new-fangled functionality. But it should get you up to speed quickly.

The drawing API works like a virtual pen that you direct, instructing it to draw lines or curves to specific coordinate positions and sometimes filling the drawn lines with color or bitmap fills. All drawing occurs through a Graphics instance and its methods, and Graphics instances are only found within a Sprite or Shape—a Graphics instance is not something

you need (or, in fact, can) ever instantiate, but it can be accessed through Sprite or Shape's `graphics` property. That means if you want to draw using the API, you must first have a Sprite or a Shape, and that instance must be added to the display list. Then all drawing must occur within that Sprite or Shape through its `graphics` property.

```
var sprite:Sprite = new Sprite();
sprite.graphics.moveTo(50, 50);
```

In the case of a Sprite, which can contain other display objects, anything you draw within it using the drawing API will be rendered below any children that the Sprite may have.

> *Anything that should be visible in a Flash movie, game, or application must be added to the display list of the Flash Player. This is a hierarchical list of all the objects currently renderable, at the root of which is the aptly named root display object. When you create a Sprite or MovieClip document class in a Flash or an ActionScript project, the root is that same document class, and it is automatically in the display list (if you are using the Flex framework, the main application class is added to the display list automatically as well, and the root display object is accessible through its `root` property).*
>
> *The display list can hold any object that is an instance of DisplayObject, an abstract class that is extended by concrete child classes like Bitmap, TextField, Video, Shape, and Sprite (which MovieClip extends). Once you instantiate an instance of one of these classes, it must be explicitly added to the display list or it will not be rendered. This is done by adding the instance as a child to a display object container that is already in the display list using the DisplayObjectContainer (which itself extends DisplayObject) methods `addChild()` or `addChildAt()`.*
>
> *As there is a lot to cover in this book for generating and manipulating graphics, we jump right in here in Chapter 1 with the ActionScript code features for doing so. To fit as much as possible into the text, some foundation features of ActionScript and the Flash Player, like using the display list, are not covered past this little focus point. For more in-depth coverage, please see Adobe's ActionScript 3.0 documentation, or a book like* Foundation ActionScript 3.0 with Flash CS3 and Flex *(ISBN: 978-1-59059-815-3) from friends of ED, which I also contributed to (and the chapter on the display list as well!).*

Once you begin drawing, the virtual pen is always at a specific coordinate position within the Sprite or Shape. It starts at (0, 0) in the display object's coordinate space and is moved around through the drawing API commands. Let's look at the basic drawing commands and explore how they can be used to manipulate that virtual pen.

Reviewing the original eight

It all began with eight little drawing commands. The following sections break them down. There won't be a quiz afterward, but it's not a bad idea to pretend there will be.

Drawing straight lines

The following `moveTo()` method places the virtual pen at a new coordinate position without drawing anything:

```
moveTo(x:Number,y:Number):void
```

It is as if the pen is picked up off of the piece of paper and set down at its new position by magic (or your hand, which could be magic, I guess). This is necessary when you need to draw a noncontinuous line or start at a coordinate position other than the default (0, 0).

```
lineTo(x:Number, y:Number):void
```

The `lineTo()` method is used to draw a line from the pen's current coordinate position to a new coordinate position specified in the parameters. This will produce a straight line from point to point.

The `lineStyle()` method is used to specify the various visual properties of a line, like its thickness and color.

```
lineStyle(
  thickness:Number=NaN,
  color:uint=0,
  alpha:Number=1.0,
  pixelHinting:Boolean=false,
  scaleMode:String="normal",
  caps:String=null,
  joints:String=null,
  miterLimit:Number=3
):void
```

There are lots of parameters with this method, so let's break them down:

- `thickness`: This specifies the thickness of the line to be drawn. A value of 0 creates a hairline stroke that doesn't scale.
- `color`: This parameter specifies the color for the line to be drawn.
- `alpha`: And this one describes the opacity for the line to be drawn.
- `pixelHinting`: Use the `pixelHinting` parameter to specify whether lines and points should be drawn at full pixel positions and at full pixel widths (`true`) or whether fractions of full pixels can be used, such as using a line thickness of 0.5 and placing an anchor point at (1.5, 1.5).
- `scaleMode`: This one specifies the way that strokes will be scaled when the display object in which they're drawn is scaled. This should be a value represented by one of the constants in the LineScaleMode class: `NORMAL`, `NONE`, `VERTICAL`, or `HORIZONTAL`.

NORMAL always scales. NONE never scales. HORIZONTAL and VERTICAL do not scale the stroke if the display object is scaled only vertically or horizontally, respectively.

- caps: The types of end caps to use on the stroke are specified with this parameter. This should be a value represented by one of the constants of CapsStyle: NONE (no cap extended off the line), ROUND (round extension of end of line), and SQUARE (square extension of end of line).
- joints: This parameter designates the type of joint to use on the stroke at any corner. This should be a value represented by one of the constants of JointStyle: MITER (creates a point only if the point's size is under the miter limit and otherwise squares it off), ROUND (rounds the corner), or BEVEL (squares off the corner). Joints and caps options are demonstrated in Figure 1-1.

Figure 1-1.
Three strokes with caps and joints set to ROUND on the left, SQUARE and BEVEL in the center, and NONE and MITER on the right

- miterLimit: This parameters works with the MITER joint style to specify how far a point at a corner of the stroke will extend before it is squared off. Figure 1-2 shows the same angle corner with three different miter limits.

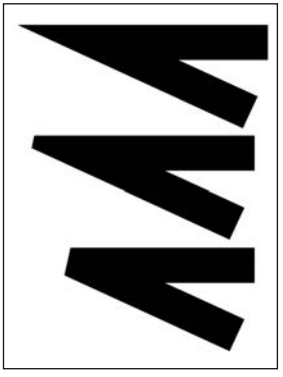

Figure 1-2.
The same stroke with three different miter limits. The top has a miter limit of 5; the middle has a miter limit of 3, and the bottom has a miter limit of 0.

Before a line can be drawn, you must specify how it should look, and that is where the lineStyle() method comes in. This method needs to be called prior to any calls to lineTo() or curveTo() or else the lines drawn will have no thickness and therefore will not be visible. When the lineStyle() was introduced, only the first three parameters were available, and these controlled the pixel thickness, color, and opacity of the lines drawn. The additional parameters were added in Flash 8 and control whether lines snap to exact pixels, how they scale when a parent display object is resized, and how their end caps and joints are rendered.

```
clear():void
```

If you need to clear previously drawn graphics, the `clear()` method not only removes all drawn lines and fills but also resets the coordinate position of the virtual pen to (0, 0) and clears the line style.

To test these four methods, you will find the following code, fully commented, in the file `DrawingStraightLines.as` in this chapter's files. You can test the compiled SWF included or, to compile this class, refer to the instructions in the appendix for how to set up and compile ActionScript projects in either Flex Builder or in Flash. Run the SWF, and click and drag multiple times on the stage. You should see something like Figure 1-3. The following class, with lines to focus on in bold, produces the SWF:

```
package {

  import flash.display.Shape;
  import flash.display.Sprite;
  import flash.events.MouseEvent;
  import flash.geom.Point;

  [SWF(width=550, height=400, backgroundColor=0xFFFFFF)]

    private var _currentShape:Shape;
    private var _color:uint;
    private var _startPosition:Point;

  public class DrawingStraightLines extends Sprite {

    public function DrawingStraightLines() {
      stage.addEventListener(MouseEvent.MOUSE_DOWN, onStageMouseDown);
      stage.addEventListener(MouseEvent.MOUSE_UP, onStageMouseUp);
    }

    private function drawLine():void {
      _currentShape.graphics.clear();
      _currentShape.graphics.lineStyle(3, _color);
      _currentShape.graphics.moveTo(_startPosition.x, ➥
_startPosition.y);
      _currentShape.graphics.lineTo(stage.mouseX, stage.mouseY);
    }

    private function onStageMouseDown(event:MouseEvent):void {
      _color = Math.random()*0xFFFFFF;
      _currentShape = new Shape();
      addChild(_currentShape);
      _startPosition = new Point(stage.mouseX, stage.mouseY);
      stage.addEventListener(MouseEvent.MOUSE_MOVE, onStageMouseMove);
    }
```

```
        private function onStageMouseUp(event:MouseEvent):void {
          stage.removeEventListener(MouseEvent.MOUSE_MOVE, ➥
    onStageMouseMove);
        }

        private function onStageMouseMove(event:MouseEvent):void {
          drawLine();
          event.updateAfterEvent();
        }

      }

    }
```

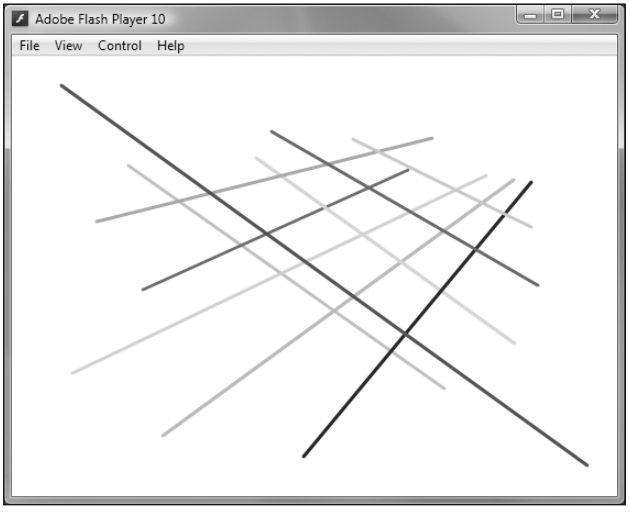

Figure 1-3. The DrawingStraightLines test that allows for dragging out randomly colored straight lines

Within this class, I set up a listener for when the stage is clicked and when the mouse is released after clicking. When the stage is clicked, the `onStageMouseDown()` handler is called. There, a random color is selected, and a new Shape is instantiated and added as a child to the display list. The coordinate position where the mouse is clicked is saved into the property `_startPosition`. Finally, a new listener is set up for when the mouse is moved, `onStageMouseMove()`.

In `onStageMouseMove()`, the custom `drawLine()` method is invoked, and the screen is immediately updated using the `updateAfterEvent()` method of the `MouseEvent` instance. I do this so that the drawing done isn't reliant on the frame rate of the application but will instead be updated whenever the mouse is moved. That drawing is of course handled in `drawLine()`, where I clear any previously drawn lines in the current shape, assign a line

style of 3-pixel thickness and the random color, move the virtual pen to the start position where the stage was clicked, and then draw a straight line to the current mouse position. I continue doing this every time the mouse is moved until the user releases the mouse and the MOUSE_MOVE listener is removed in the onStageMouseUp() handler.

It's a very simple drawing application done in a small amount of code. Take that, pre-rendered graphics!

Drawing curves

When you need a curve between two points, you would use the following method:

```
curveTo(
  controlX:Number,
  controlY:Number,
  anchorX:Number,
  anchorY:Number
):void
```

This draws a curved line between the current coordinates of the pen with the new coordinates specified with the anchorX and anchorY parameters. The controlX and controlY parameters are the position of the control point used to specify how the line curves using a quadratic Bezier equation.

Now, what the heck is meant by "a quadratic Bezier equation"? If you are familiar with a program like Illustrator, you will be used to Bezier curves drawn using a cubic equation, which uses two control handles to define how a curve is drawn between two anchors. The quadratic Bezier curve uses only one control point. Conceptually, you can imagine that the control point is like a magnet that is pulling at the line—a drawn line will never go directly through the control point but will be pulled in its direction.

To test and visualize this, have a look at DrawingCurves.as in this chapter's files. If you compile this class (remember, you can refer to the appendix for instructions) or test the SWF directly, you should see the result shown in Figure 1-4, which shows a graphic representation of the two anchor points of a curve and its control point. You can drag any of the points around to alter the curve. This is produced using the following code, with the drawing code set in bold for you:

```
package {

  import flash.display.Sprite;
  import flash.events.MouseEvent;

  [SWF(width=550, height=400, backgroundColor=0xFFFFFF)]

  public class DrawingCurves extends Sprite {

    private var _controlPoint:Sprite;
    private var _anchor0:Sprite;
    private var _anchor1:Sprite;
```

```
    public function DrawingCurves() {
      _anchor0 = addControlPoint(50, 300);
      _anchor1 = addControlPoint(500, 300);
      _controlPoint = addControlPoint(275, 100);
      drawCurve();
    }

    private function addControlPoint(x:Number, y:Number):Sprite {
      var controlPoint:Sprite = new Sprite();
      controlPoint.graphics.lineStyle(20);
      controlPoint.graphics.lineTo(1, 0);
      controlPoint.addEventListener(MouseEvent.MOUSE_DOWN, ➥
onControlDown);
      controlPoint.addEventListener(MouseEvent.MOUSE_UP, onControlUp);
      controlPoint.x = x;
      controlPoint.y = y;
      addChild(controlPoint);
      return controlPoint;
    }

    private function drawCurve():void {
      graphics.clear();
      graphics.lineStyle(3, 0xFF);
      graphics.moveTo(_anchor0.x, _anchor0.y);
      graphics.curveTo(_controlPoint.x, _controlPoint.y, ➥
_anchor1.x, _anchor1.y);
      graphics.lineStyle(1, 0, .5);
      graphics.lineTo(_controlPoint.x, _controlPoint.y);
      graphics.lineTo(_anchor0.x, _anchor0.y);
    }

    private function onControlDown(event:MouseEvent):void {
      (event.target as Sprite).startDrag();
      stage.addEventListener(MouseEvent.MOUSE_MOVE, onControlMove);
    }

    private function onControlUp(event:MouseEvent):void {
      (event.target as Sprite).stopDrag();
      stage.removeEventListener(MouseEvent.MOUSE_MOVE, onControlMove);
    }

    private function onControlMove(event:MouseEvent):void {
      drawCurve();
      event.updateAfterEvent();
    }

  }

}
```

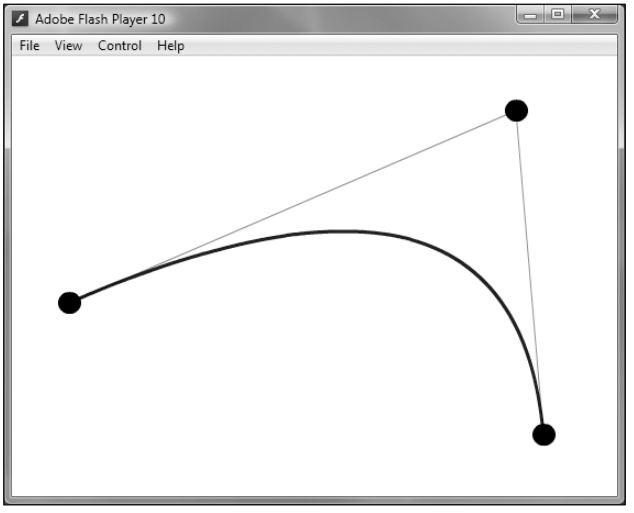

Figure 1-4. The DrawingCurves test showing how a control point acts with anchor points to make a quadratic Bezier curve

In this class, three sprites are created to represent the two anchors and control point. These are created in the `addControlPoint()` method, which has a simple trick of using a very thick line style (20 pixels) with a very short line in order to create a circle.

```
controlPoint.graphics.lineStyle(20);
controlPoint.graphics.lineTo(1, 0);
```

The main drawing occurs in `drawCurve()`, which is called whenever any of the points is dragged around the stage. Within the method, I first clear any previously drawn graphics and then create a 3-pixel thick blue line for the curve, which is drawn from `_anchor0` to `_anchor1` using first `moveTo()` and then `curveTo()`. I next change the line style to a thin transparent black line and draw a straight line from the current pen position at the second anchor to the control point, then another straight line to the first anchor to show how the two anchor points connect with the control point to define the curve.

Drawing solid fills

If you wish to fill a shape with a solid color, the following method is available to you:

```
beginFill(color:uint, alpha:Number=1.0):void
```

This method can be called prior to any calls to `lineTo()` or `curveTo()`, and the shape that is formed by these drawing methods will be filled with the solid color specified.

The other bookend for `beginFill()` is the following method:

```
endFill():void
```

This method should be called at the completion of the drawing commands that included an initial instruction to draw a fill (like `beginFill()` or `beginGradientFill()`) and instructs the Flash Player to render the fill.

Have a look at DrawingSolidFills.as to see solid fills in action. If you test this class, you will see a new background fill drawn for the entire movie whenever the stage is clicked. The following code accomplishes this, with the relevant drawing API lines in bold:

```
package {

 import flash.display.Sprite;
 import flash.events.MouseEvent;

 [SWF(width=550, height=400, backgroundColor=0xFFFFFF)]

 public class DrawingSolidFills extends Sprite {

   public function DrawingSolidFills() {
     stage.addEventListener(MouseEvent.CLICK, onStageClick);
     drawBackground();
   }

   private function drawBackground():void {
     graphics.clear();
     graphics.beginFill(Math.random()*0xFFFFFF);
     graphics.lineTo(stage.stageWidth, 0);
     graphics.lineTo(stage.stageWidth, stage.stageHeight);
     graphics.lineTo(0, stage.stageHeight);
     graphics.lineTo(0, 0);
     graphics.endFill();
   }

   private function onStageClick(event:MouseEvent):void {
     drawBackground();
   }

 }

}
```

In a later section, we will discuss drawRect(), which would simplify this drawing code (which, admittedly, is already fairly simple). In this case, I draw a rectangle covering the entire stage manually using four lineTo() commands. In the beginFill() call, I pass a random color selected from the 16 million or so available.

Drawing gradient fills

The following method allows you to fill a shape with a gradient of colors and/or alphas:

```
beginGradientFill(
  type:String,
  colors:Array,
  alphas:Array,
  ratios:Array,
```

```
    matrix:Matrix=null,
    spreadMethod:String="pad",
    interpolationMethod:String="rgb",
    focalPointRatio:Number=0
):void
```

That's a complex little method. Let's break down each of the parameters.

- `type`: The type of gradient to draw should be either `GradientType.LINEAR` or `GradientType.RADIAL`. Linear gradients are bands drawn in a straight line between points, while radial gradients are drawn in rings out from a central point.
- `colors`: This specifies an array of colors to be used in the gradient. For a linear gradient, these will be the colors from left to right (assuming you haven't rotated the gradient). For a radial gradient, the colors will be from the center out.
- `alphas`: An array of alpha values to be used in the gradient, with values from 0 to 1, can be specified with this parameter. There must be the same number of alpha values as there are colors, which each index in the `alphas` array corresponding to the alpha value of the color at the same index in the `colors` array.
- `ratios`: This holds an array of values that specify where each color in the `colors` array is distributed on the length of the full gradient. Each index should hold a value between 0 and 255, with 0 being the left (`LINEAR`) or center (`RADIAL`) of the gradient and 255 being the right (`LINEAR`) or outer radius (`RADIAL`) of the gradient. Just as with `alphas`, the length of this array must match the length of the colors and each index corresponds with the color value of the same index in the `colors` array.

  As an example of how ratios are used, imagine you have a rectangle that is 100 pixels wide that you fill with a linear gradient. You specify three colors, red, green, and blue, with the ratios 0, 128, and 255, respectively. This would place the full red on the left of the rectangle, the full green pretty much in the center, and the full blue on the right, with gradient values between each color. Now, if you changed the ratios to 0, 64, and 128, the full green would be at around a quarter of the width of the rectangle with the full blue at the center and extending to the right edge. Figure 1-5 shows these two scenarios.

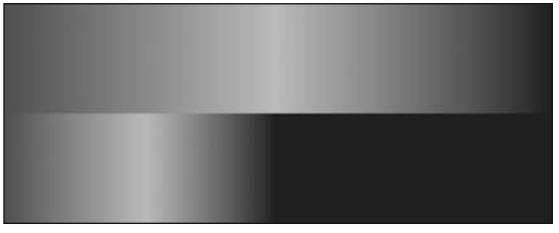

Figure 1-5. The same gradient colors applied to the same dimensional shape with two different ratio values to control color distribution on the gradient

- `matrix`: This transformation matrix determines how the gradient will be moved, scaled, and/or rotated within its drawn shape. This must be an instance of the flash.geom.Matrix class. We'll get into matrices next chapter, but for gradients, you generally only have to create an instance and use its handy `createGradientBox()` method, which we'll look at in the next example.

- spreadMethod: This parameter determines how a gradient that is smaller than the width or height of the drawn shape will extend to fill the shape. This should be a value as represented by the constants of the SpreadMethod class: PAD, REFLECT, and REPEAT. PAD simply continues with the end color of the gradient. REFLECT reverses the gradient. REPEAT, rather unsurprisingly, repeats the gradient. Each of these spread methods is shown in Figure 1-6.

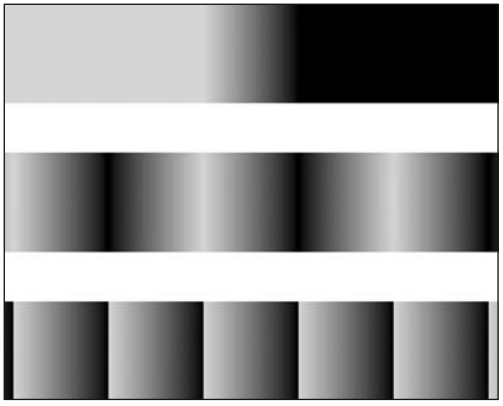

Figure 1-6.
The same gradient colors applied to the same dimensional object, with the top object using a PAD spread method, the middle using REFLECT, and the bottom using REPEAT

- interpolationMethod: Specify how intermediate colors in a gradient are calculated with this parameter. This should be a value represented by a constant found in InterpolationMethod: LINEAR_RGB or RGB.
- focalPointRatio: For radial gradients, this determines where the focal point of the gradient lies along its horizontal diameter. –1.0 and 1.0 place the focal point on the radius of the gradient, and 0 places the focal point in the center. Any intermediate value places it in between. Figure 1-7 shows several examples of the focal point adjusted for the same radial gradient.

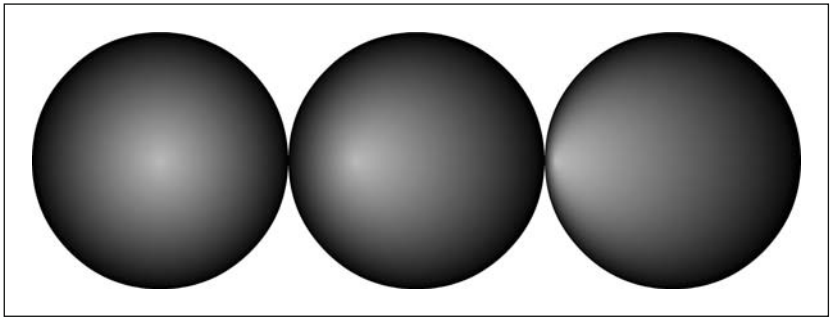

Figure 1-7. Three shapes using the same radial gradient colors and center, but with the focal point increasingly shifted to the left

If you would rather fill a drawn shape with a gradient color as opposed to a solid color, then the beginGradientFill() method is for you. Its type can either be linear or radial, and you must at least include an array of colors, their alphas, and their ratio positions to use. The matrix is optional (but often necessary) to rotate, reposition, or resize the gradient