# AdvancED ActionScript 3.0 Animation

Keith Peters

# AdvancED ActionScript 3.0 Animation



ISBN-13 (pbk): 978-1-4302-1608-7

ISBN-13 (electronic): 978-1-4302-1608-7

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 Reviewer**
Seb Lee-Delisle

**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**
Sofia Marchant

**Copy Editor**
Nancy Sixsmith

**Associate Production Director**
Kari Brooks-Copony

**Production Editor**
Janet Vail

**Compositor**
Lynn L'Heureux

**Proofreader**
Nancy Bell

**Indexer**
Carol Burbo

**Artist**
Kinetic Publishing Services, LLC

**Cover Image Designer**
Bruce Tang

**Interior and Cover Designer**
Kurt Krames

**Manufacturing Director**
Tom Debolski

*To Miranda and Kristine, for their patience and support, once again.*

# CONTENTS AT A GLANCE

# CONTENTS

# ABOUT THE AUTHOR

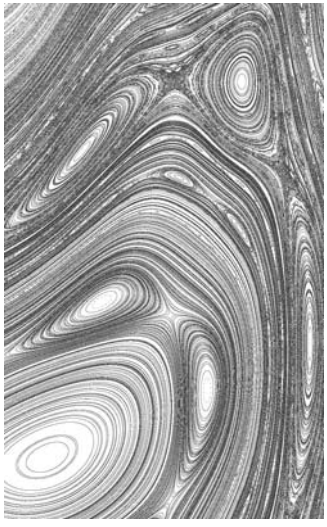

**Keith Peters** is a non-recovering Flash addict, author of several books on Flash and ActionScript, speaker at Flash conferences around the world, and owner of various Flash-related web sites (`www.bit-101.com`, `www.artfromcode.com`, and `www.wickedpissahgames.com`).

Keith lives in Wellesley, Massachusetts with his wife Miranda and daughter Kristine, in a house that Flash helped pay for. He works as a senior Flash programmer at Infrared5 in Boston.

# ABOUT THE TECHNICAL REVIEWER

**Seb Lee-Delisle** has been working in digital media for more than 15 years and is one of the founding partners of UK Flash specialists Plug-in Media (`http://pluginmedia.net`), working with clients such as BBC, Sony, Philips, Unilever, and Barclays. He is also one of the developers of Papervision3D, the highly successful open source, real time 3D ActionScript library. Seb's work with Plug-in Media has pushed the boundaries of 3D and gaming in Flash. He has recently completed the live 3D GameDay visualizations for Major League Baseball and a real time 3D website for the BBC kids' show Big and Small.

# ABOUT THE COVER IMAGE DESIGNER



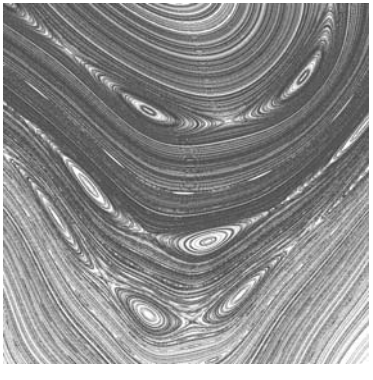
**Bruce Tang** is a freelance web designer, visual programmer, and author from Hong Kong. His main creative interest is generating stunning visual effects using Flash or Processing.

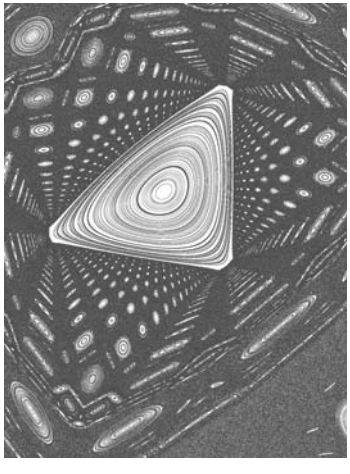
Bruce has been an avid Flash user since Flash 4, when he began using Flash to create games, websites, and other multimedia content. After several years of ActionScripting, he found himself increasingly drawn toward visual programming and computational art. He likes to integrate math and physics into his work, simulating 3D and other real-life experiences onscreen. His first Flash book was published in October 2005. Bruce's folio, featuring Flash and Processing pieces, can be found at `www.betaruce.com`, and his blog at `www.betaruce.com/blog`.

The cover image uses a high-resolution Henon phase diagram generated by Bruce with Processing, which he feels is an ideal tool for such experiments. Henon is a strange attractor created by iterating through some equations to calculate the coordinates of millions of points. The points are then plotted with an assigned color.

$$x_{n+1} = x_n \cos(a) - (y_n - x_n^p) \sin(a)$$

$$y_{n+1} = x_n \sin(a) + (y_n - x_n^p) \cos(a)$$





# ACKNOWLEDGMENTS

Little—if any—of the material in this book is stuff I dreamed up in my own head. Thanks to the hundreds of programmers, developers, scientists, mathematicians, and physicists who studied, researched, programmed, translated, and made their work available for others to benefit from.

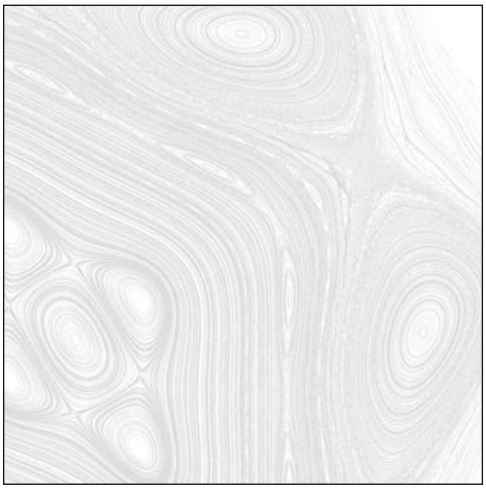
## Layout conventions

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

- Important words or concepts are normally highlighted on the first appearance in *italics*.
- Code is presented in `fixed-width font`.
- New or changed code is normally presented in **`bold fixed-width font`**.
- Pseudo-code and variable input are written in *`italic 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 we 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

# ADVANCED COLLISION DETECTION

Collision detection is the math, art, science, or general guesswork used to determine whether some object has hit another object. This sounds pretty simple, but when you are dealing with objects that exist only in a computer's memory and are represented by a collection of various properties, some complexities can arrive.

The basic methods of collision detection are covered in *Foundation ActionScript 3.0 Animation: Making Things Move!* (hereafter referred to as *Making Things Move*). This chapter looks at one method of collision detection that wasn't covered in that book and a strategy to handle collisions between large amounts of objects.

Note that the subject of collision detection does not delve into what you do *after* you detect a collision. If you are making a game, you might want the colliding objects to blow up, change color, or simply disappear. One rather complex method of handling the results of a collision was covered in the "Conservation of Momentum" chapter of *Making Things Move*. But ultimately it's up to you (and the specs of the application or game you are building) to determine how to respond when a collision is detected.

# Hit Testing Irregularly Shaped Objects

*Making Things Move* covered a few basic methods of collision detection, including the built-in `hitTestObject` and `hitTestPoint` methods, as well as distance-based collision detection. Each of these methods has its uses in terms of the shapes of objects on which you are doing collision detection. The `hitTestObject` method is great for detecting collisions between two rectangular-shaped objects, but will often generate false positives for other shapes. The `hitTestPoint` method is suitable for finding out whether the mouse is over a particular object or whether a very small point-like object has collided with any other shaped object, but it is rather useless for two larger objects. Distance-based collision detection is great for circular objects, but will often miss collisions on other shaped objects.

The Holy Grail of collision detection in Flash has been to test two irregularly shaped objects against each other and accurately know whether or not they are touching. Although it wasn't covered in *Making Things Move*, a method has existed for doing this via the `BitmapData` class since Flash 8. In fact, the method is even called `hitTest`.

First, a note on terminology. ActionScript contains a `BitmapData` class, which holds the actual bitmap image being displayed, and a `Bitmap` class, which is a display object that contains a `BitmapData` and allows it to be added to the display list. If I am referring to either one of these classes specifically, or an instance of either class, I will use the capitalized version. But often I might casually use the term **bitmap** in lowercase to more informally refer to a bitmap image. Do not confuse it with the `Bitmap` class.

`BitmapData.hitTest` compares two `BitmapData` objects and tells you whether any of their pixels are overlapping. Again, this sounds simple, but complexities arise once you start to think about it. Bitmaps are rectangular grids of pixels, so taken in its simplest form, this method would be no more complex (or useful) than the `hitTestObject` method on a display object. Where it really starts to get useful is when you have a transparent bitmap with a shape drawn in it.

When you create a `BitmapData` object, you specify whether it will support transparency right in the constructor:

```
new BitmapData(width, height, transparent, color);
```

That third parameter is a Boolean value (`true`/`false`) that sets the transparency option. If you set it to `false`, the bitmap will be completely opaque. Initially, it will appear as a rectangle filled with the specified background color. You can use the various `BitmapData` methods to change any of the pixels in the bitmap, but they will always be fully opaque and cover anything behind that `BitmapData`. Color values for each pixel will be 24-bit numbers in the form 0xRRGGBB. This is a 6-digit hexadecimal number, where the first pair of numbers specifies the value for the red channel from 00 (0) to FF (255), the second pair sets the green channel, and the third sets the blue channel. For example, 0xFFFFFF would be white, 0xFF0000 would be red, and 0xFF9900 would be orange. For setting and getting values of individual pixels, you would use the methods `setPixel` and `getPixel`, which use 24-bit color values.

However, when you specify `true` for the transparency option in a `BitmapData` class, each pixel now supports an alpha channel, using a 32-bit number in the format 0xAARRGGBB. Here, the first 2 digits represent the level of transparency for a pixel, where 00 would be completely transparent, and FF would be fully opaque. In a transparent `BitmapData`, you would use `setPixel32` and `getPixel32` to set and read colors of individual pixels. These methods take 32-bit numbers. Note that if you pass

in a 24-bit number to one of these methods, the alpha channel will be evaluated as being 0, or fully transparent.

To see the exact difference between the two, let's create one of each. You can use the following class as the main class in a Flex Builder 3 or 4 ActionScript Project, or as the document class in Flash CS3 or CS4. This class, `BitmapCompare`, is available at this book's download site at `www.friendsofed.com`.

```
package
{
    import flash.display.Bitmap;
    import flash.display.BitmapData;
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.geom.Rectangle;

    public class BitmapCompare extends Sprite
    {
        public function BitmapCompare()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;

            // draw a bunch of random lines
            graphics.lineStyle(0);
            for(var i:int = 0; i < 100; i++)
            {
                graphics.lineTo(Math.random() * 300,
                                Math.random() * 400);
            }

            // create an opaque bitmap
            var bmpd1:BitmapData = new BitmapData(300, 200,
                                                  false, 0xffffff);
            bmpd1.fillRect(new Rectangle(100, 50, 100, 100), 0xff0000);
            var bmp1:Bitmap = new Bitmap(bmpd1);
            addChild(bmp1);

            // create a transparent bitmap
            var bmpd2:BitmapData = new BitmapData(300, 200,
                                                  true, 0x00ffffff);
            bmpd2.fillRect(new Rectangle(100, 50, 100, 100),
                           0xffff0000);
            var bmp2:Bitmap = new Bitmap(bmpd2);
            bmp2.y = 200;
            addChild(bmp2);
        }
    }
}
```

This code first draws a bunch of random lines on the stage, just so you can tell the difference between the stage and the bitmaps. It then creates two bitmaps and draws red squares in the center of each. The top bitmap is opaque and covers the lines completely. The bottom bitmap is transparent, so only the red square covers the lines on the stage. You can see the result in Figure 1-1.

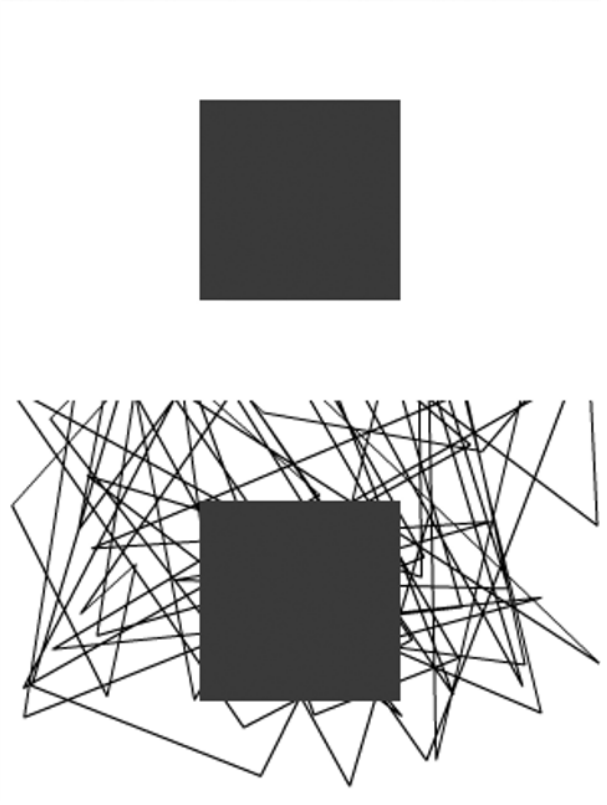


**Figure 1-1.** An opaque bitmap on top, transparent below

Furthermore, with a transparent bitmap you can use partial transparency. Change the second `fillRect` statement in the last code sample to the following:

```
bmpd2.fillRect(new Rectangle(100, 50, 100, 100), 0x80FF0000);
```

Note that we used a 32-bit AARRGGBB color value for the fill, and the alpha value has been halved to 0x80, or 128 in decimal. This makes the red square semitransparent, as seen in Figure 1-2.

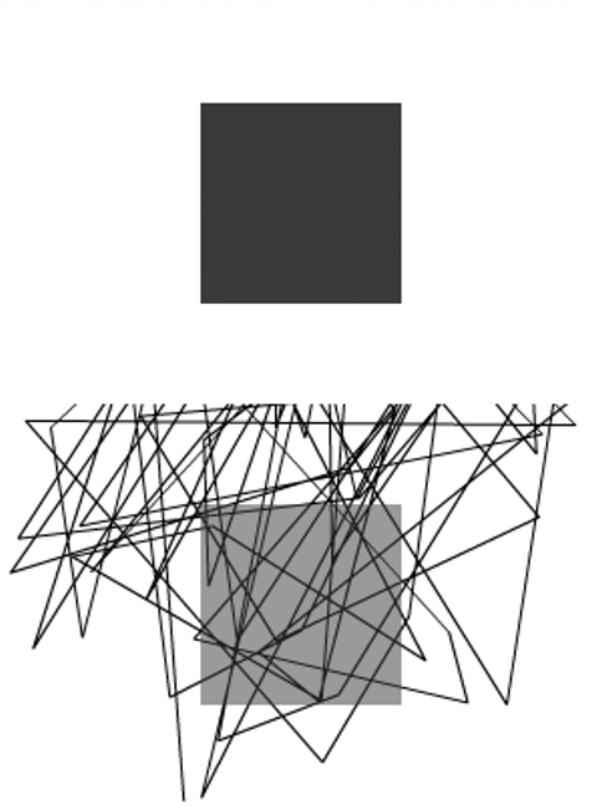


**Figure 1-2.** A semitransparent square

## Bitmaps for collision detection

So now let's take a look at how to use bitmaps to achieve collision detection. First, we'll need a nice irregular shape to test with. A five-pointed star will do nicely. Why not make it into its very own class so we can reuse it? Here's the `Star` class, also available at the book's download site:

```
package
{
    import flash.display.Sprite;

    public class Star extends Sprite
    {
        public function Star(radius:Number, color:uint = 0xFFFF00):void
        {
            graphics.lineStyle(0);
            graphics.moveTo(radius, 0);
```

```
                graphics.beginFill(color);
                // draw 10 lines
                for(var i:int = 1; i < 11; i++)
                {
                    var radius2:Number = radius;
                    if(i % 2 > 0)
                    {
                     // alternate radius to make spikes every other line
                        radius2 = radius / 2;
                    }
                    var angle:Number = Math.PI * 2 / 10 * i;
                    graphics.lineTo(Math.cos(angle) * radius2,
                                    Math.sin(angle) * radius2);
                }
            }
        }
    }
```

This just draws a series of lines at increasing angles and alternate radii, which cleverly form a star. And here is the class that does the hit testing. Again, like most of the code in this book, it can be used either as a document class in Flash CS3 or CS4, or as a main application class in Flex Builder 3 or 4, and is available from the book's download site.

```
package
{
    import flash.display.Bitmap;
    import flash.display.BitmapData;
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.MouseEvent;
    import flash.filters.GlowFilter;
    import flash.geom.Matrix;
    import flash.geom.Point;

    public class BitmapCollision1 extends Sprite
    {
        private var bmpd1:BitmapData;
        private var bmp1:Bitmap;
        private var bmpd2:BitmapData;
        private var bmp2:Bitmap;

        public function BitmapCollision1()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;

            // make a star
            var star:Star = new Star(50);
```

```
            // make a fixed bitmap, draw the star into it
            bmpd1 = new BitmapData(100, 100, true, 0);
            bmpd1.draw(star, new Matrix(1, 0, 0, 1, 50, 50));
            bmp1 = new Bitmap(bmpd1);
            bmp1.x = 200;
            bmp1.y = 200;
            addChild(bmp1);

            // make a moveable bitmap, draw the star into it, too
            bmpd2 = new BitmapData(100, 100, true, 0);
            bmpd2.draw(star, new Matrix(1, 0, 0, 1, 50, 50));
            bmp2 = new Bitmap(bmpd2);
            addChild(bmp2);

            stage.addEventListener(MouseEvent.MOUSE_MOVE,
                                   onMouseMoving);
        }

        private function onMouseMoving(event:MouseEvent):void
        {
            // move bmp2 to the mouse position (centered).
            bmp2.x = mouseX - 50;
            bmp2.y = mouseY - 50;

            // the hit test itself.
            if(bmpd1.hitTest(new Point(bmp1.x, bmp1.y), 255, bmpd2,
                             new Point(bmp2.x, bmp2.y), 255))
            {
                bmp1.filters = [new GlowFilter()];
                bmp2.filters = [new GlowFilter()];
            }
            else
            {
                bmp1.filters = [];
                bmp2.filters = [];
            }
        }
    }
}
```

Here we create a star using the `Star` class and draw it into two bitmaps. We use a matrix to offset the star during drawing by 50 pixels on each axis because the registration point of the star is in its center, and the registration point of the bitmap is at the top left. We offset it so we can see the whole star.

One of these bitmaps (bmp1) is in a fixed position on the stage; the other (bmp2) is set to follow the mouse around. The key line comes here:

```
if(bmpd1.hitTest(new Point(bmp1.x, bmp1.y), 255, bmpd2,
                 new Point(bmp2.x, bmp2.y), 255))
```

This is what actually determines if the two bitmaps are touching. The signature for the BitmapData.hitTest method looks like this:

```
hitTest(firstPoint:Point,
        firstAlphaThreshold:uint,
        secondObject:Object,
        secondPoint:Point,
        secondAlphaThreshold:uint);
```

You'll notice that the parameters are broken down into two groups: first and second. You supply a point value for each. This corresponds to the top-left corner of BitmapData. The reason for doing this is that each bitmap might be nested within another symbol or deeply nested within multiple symbols. In such a case, they might be in totally different coordinate systems. Specifying an arbitrary point lets you align the two coordinate systems if necessary, perhaps through using the DisplayObject.localToGlobal method. In this example, however, both bitmaps will be right on the stage, so we can use their local position directly to construct the point for each.

The next first/last parameters are for the alpha threshold. As you saw earlier, in a transparent BitmapData, each pixel's transparency can range from 0 (fully transparent) to 255 (fully opaque). The alpha threshold parameters specify how opaque a pixel must be in order to register a hit. In this example, we set both of these to 255, meaning that for a pixel in either bitmap to be considered for a hit test, it must be fully opaque. We'll do another example later that shows the use of a lower threshold.

Finally, there is the secondObject parameter. Note that it is typed to an object. Here you can use a Point, a Rectangle, or another BitmapData as the object to test against. If you are using a Point or Rectangle, you do not need to use the final two parameters. Testing against a Point is useful if you want to test whether the mouse is touching a bitmap. A quick example follows:

```
if(myBitmapData.hitTest(new Point(myBitmapData.x, myBitmapData.y),
                        255,
                        new Point(mouseX, mouseY)))
{
    // mouse is touching bitmap
}
```

I can't think of a particularly useful example for testing a bitmap against a rectangle, but it's good to know that if the need arises, it's there!

In our example, however, we are using another BitmapData object, so we pass that in along with the second Point and alpha threshold.

Finally, if there is a hit, we give each star a red glow through the use of a default glow filter. If no hit, we remove any filter. You can see the results in Figures 1-3 and 1-4.

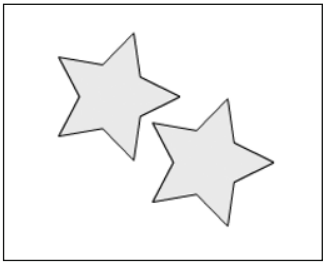


**Figure 1-3.** Stars are not touching.

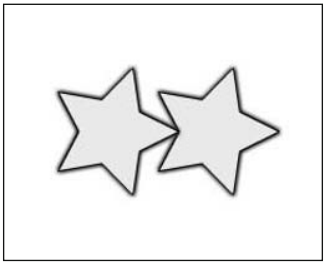


**Figure 1-4.** And now they are.

Play with this for awhile, and you'll see that it truly is pixel-to-pixel collision detection.

## Hit testing with semitransparent shapes

In the preceding example, we drew a star that was totally opaque into each bitmap. We were thus testing against fully opaque pixels in each bitmap and therefore we set the alpha threshold to 255 in each one. (We actually could have set the alpha threshold to anything above zero and had the same effect.)

Now let's look at hit testing with a shape that *isn't* fully opaque. We'll alter the `BitmapCollsion1` class slightly, naming it `BitmapCollision2` (available for download on the book's site):

```
package
{
    import flash.display.Bitmap;
    import flash.display.BitmapData;
    import flash.display.GradientType;
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.MouseEvent;
    import flash.filters.GlowFilter;
    import flash.geom.Matrix;
    import flash.geom.Point;

    public class BitmapCollision2 extends Sprite
    {
        private var bmpd1:BitmapData;
        private var bmp1:Bitmap;
        private var bmpd2:BitmapData;
        private var bmp2:Bitmap;
```

```
public function BitmapCollision2()
{
    stage.align = StageAlign.TOP_LEFT;
    stage.scaleMode = StageScaleMode.NO_SCALE;

    // make a star
    var star:Star = new Star(50);

    // make a gradient circle
    var matrix:Matrix = new Matrix();
    matrix.createGradientBox(100, 100, 0, -50, -50);
    var circle:Sprite = new Sprite();
    circle.graphics.beginGradientFill(GradientType.RADIAL,
                                      [0, 0], [1, 0]
                                      [0, 255], matrix);
    circle.graphics.drawCircle(0, 0, 50);
    circle.graphics.endFill();

    // make a fixed bitmap, draw the star into it
    bmpd1 = new BitmapData(100, 100, true, 0);
    bmpd1.draw(star, new Matrix(1, 0, 0, 1, 50, 50));
    bmp1 = new Bitmap(bmpd1);
    bmp1.x = 200;
    bmp1.y = 200;
    addChild(bmp1);

    // make a moveable bitmap, draw the star into it, too
    bmpd2 = new BitmapData(100, 100, true, 0);
    bmpd2.draw(circle, new Matrix(1, 0, 0, 1, 50, 50));
    bmp2 = new Bitmap(bmpd2);
    addChild(bmp2);

    stage.addEventListener(MouseEvent.MOUSE_MOVE,
                           onMouseMoving);
}

private function onMouseMoving(event:MouseEvent):void
{
    // move bmp2 to the mouse position (centered).
    bmp2.x = mouseX - 50;
    bmp2.y = mouseY - 50;

    // the hit test itself.
    if(bmpd1.hitTest(new Point(bmp1.x, bmp1.y), 255, bmpd2,
                     new Point(bmp2.x, bmp2.y), 255))
    {
        bmp1.filters = [new GlowFilter()];
        bmp2.filters = [new GlowFilter()];
    }
```

```
                else
                {
                    bmp1.filters = [];
                    bmp2.filters = [];
                }
            }
        }
    }
```

Here we make a new `Sprite` named `circle` and draw a radial gradient-filled circle shape in it. We draw this to `bmpd2` instead of the star. If you test this, you'll see that no hit will be registered until the very center of the circle touches the star because only at the center is the circle fully opaque. You can see the results in Figures 1-5 and 1-6.

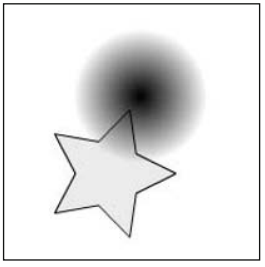


**Figure 1-5.** The star is touching the circle, but not a pixel that has the required alpha threshold.

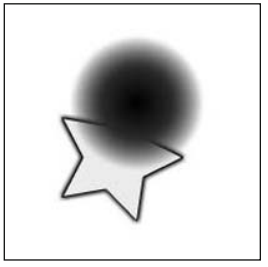


**Figure 1-6.** Only the center of the circle has an alpha of 255, so you get a hit.

Change the hit test line to make the second alpha threshold a lower value like so:

```
if(bmpd1.hitTest(new Point(bmp1.x, bmp1.y), 255, bmpd2,
                 new Point(bmp2.x, bmp2.y), 128))
```

Now you have to move the circle only part way onto the square, just so it hits a pixel whose alpha is at least 128. Try setting that second alpha threshold to different values to see the effects. Note that if you set it to zero, you might get a hit even before the circle touches the star because it will successfully hit test even against the fully transparent pixels in the very corner of the bitmap. Remember that the bitmap itself is still a rectangle, even if you can't see it all. Also note that changing the first alpha threshold (to anything other than 0) won't change anything because the star doesn't. have any semitransparent pixels—they are either fully transparent or fully opaque.

## Using BitmapData.hitTest for nonbitmaps

In the examples so far, we've been using `Bitmap` objects directly as the display objects we are moving around and testing against. But in many (if not most) cases, you'll actually be moving around different types of display objects such as `MovieClip`, `Sprite`, or `Shape` objects. Because you can't do this type of hit testing on these types of objects, you'll need to revise the setup a bit. The strategy is to keep a couple of offline `BitmapData` objects around, but not on the display list. Each time you want to check a collision between two of your actual display objects, draw one to each bitmap and perform your hit test on the bitmaps.

Realize that this is not the only way, or necessarily the best possible way, of using bitmaps for collision detection. There are probably dozens of possible methods, and this one works fine. Feel free to use it as is or improve on it.

Here's the class, `BitmapCollision3` (download it from the book's site):

```
package
{
    import flash.display.BitmapData;
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.MouseEvent;
    import flash.filters.GlowFilter;
    import flash.geom.Matrix;
    import flash.geom.Point;

    public class BitmapCollision3 extends Sprite
    {
        private var bmpd1:BitmapData;
        private var bmpd2:BitmapData;
        private var star1:Star;
        private var star2:Star;

        public function BitmapCollision3()
        {
            stage.align = StageAlign.TOP_LEFT;
            stage.scaleMode = StageScaleMode.NO_SCALE;

            // make two stars, add to stage
            star1 = new Star(50);
            addChild(star1);

            star2 = new Star(50);
            star2.x = 200;
            star2.y = 200;
            addChild(star2);

            // make two bitmaps, not on stage
            bmpd1 = new BitmapData(stage.stageWidth, stage.stageHeight, true, 0);
            bmpd2 = bmpd1.clone();

            stage.addEventListener(MouseEvent.MOUSE_MOVE,
                                   onMouseMoving);
        }
```

```
            private function onMouseMoving(event:MouseEvent):void
            {
                // move star1 to the mouse position
                star1.x = mouseX;
                star1.y = mouseY;

                // clear the bitmaps
                bmpd1.fillRect(bmpd1.rect, 0);
                bmpd2.fillRect(bmpd2.rect, 0);

                // draw one star to each bitmap
                bmpd1.draw(star1,
                           new Matrix(1, 0, 0, 1, star1.x, star1.y));
                bmpd2.draw(star2,
                           new Matrix(1, 0, 0, 1, star2.x, star2.y));

                // the hit test itself.
                if(bmpd1.hitTest(new Point(), 255, bmpd2, new Point(), 255))
                {
                    star1.filters = [new GlowFilter()];
                    star2.filters = [new GlowFilter()];
                }
                else
                {
                    star1.filters = [];
                    star2.filters = [];
                }
            }
        }
    }
```

In the constructor this time, we make two `BitmapData` objects and two stars. There's no need to put the `BitmapData` objects in `Bitmaps`, as they are not going on the display list. The stars, on the other hand, do get added to the display list. The first star, `star1`, gets moved around with the mouse. Each time the mouse is moved, both bitmaps are cleared by using `fillRect`, passing in a color value of zero. Remember that if the alpha channel is not specified, it is taken as zero, so this has the result of making all pixels completely transparent. Then each star is drawn to its corresponding bitmap:

```
bmpd1.draw(star1, new Matrix(1, 0, 0, 1, star1.x, star1.y));
bmpd2.draw(star2, new Matrix(1, 0, 0, 1, star2.x, star2.y));
```

The matrix uses the stars' x and y positions as translation values, resulting in each star being drawn in the same position it is in on the stage. Now we can do the hit test:

```
if(bmpd1.hitTest(new Point(), 255, bmpd2, new Point(), 255))
```

Because `BitmapData` is not on the display list or even in a `Bitmap` wrapper, and because both stars are in the same coordinate space and have been drawn to each `BitmapData` in their relative positions, we don't need to do any correction of coordinate spaces. We just pass in a new default `Point` (which will have x and y both zero) to each of the `Point` arguments. We'll leave the alpha thresholds at 255 because both stars are fully opaque.

Although this example doesn't look any different from the others, it's actually completely inverted, with the bitmaps invisible and the stars visible. Yet it works exactly the same way.

These are just a few examples of using `BitmapData.hitTest` to do collision detection on noncircle, rectangle, or point-shaped objects. I'm sure once you get how it all works, you can think up some cool variations for it.

Next up, we'll look at how to do collision detection on a large scale.

# Hit Testing with a Large Number of Objects

ActionScript in Flash Player 10 runs faster than ever before and it lets us do more stuff at once and move more objects at the same time. But there are still limits. If you start moving lots of objects on the screen, sooner or later things will start to bog down. Collision detection among large numbers of objects compounds the problem because each object needs to be compared against every other object. This is not limited to collision detection only; any particle system or game in which a lot of objects need to interact with each other, such as via gravity or flocking (see Chapter 2), will run into the same problems.

If you have just six objects interacting with each other, each object needs to pair up with every other object and do its hit test, gravitational attraction, or whatever action it needs to do with that other object. At first glance, this means 6 times 6, or 36 individual comparisons. But, as described in *Making Things Move*, it's actually fewer than half of that: 15 to be precise. Given objects A, B, C, D, E, F, you need to do the following pairings:

AB, AC, AD, AE, AF

BC, BD, BE, BF

CD, CE, CF

DE, DF

EF

Notice that B does not have to check with A because A has already checked with B. By the time you get to E, it's already been checked against everything but F. And after that, F has been checked by all the others. The formula for how many comparisons need to occur is as follows, where N is the number of objects:

$(N^2 – N)/2$

For 6 objects, that's (36 – 6)/2 or 15.

For 10 objects, that's (100 – 10)/2 or 45 checks.

20 objects means 190 checks, and 30 objects is 435!

You see that this goes up very quickly, and you need to do something to limit it. One hundred objects aren't really hard to move around the screen in ActionScript 3.0, but when you start doing collision detection or some other interobject comparisons, that's 4,950 separate checks to do! If you are using distance-based collision detection, that's 4,950 times calculating the distance between two objects. If you're using bitmap collision, as described earlier in the chapter, that's 4,950 times clearing two bitmaps, drawing two objects, and calling the `hitTest` method. On every frame! That's bound to slow your SWF file down.

Fortunately, there is a trick to limit the number of checks you need to do. Think about this: if two relatively small objects are on opposite sides of the screen, there's no way they could possibly be colliding. But to discover that, we need to calculate the distance between them, right? So we are back to square one. But maybe there's another way.

Suppose that we break down the screen into a grid of square cells, in which each cell is at least as large as the largest object, and then we assign each object to one of the cells in that grid—based on where the center of that object is located. If we set it up just right, an object in a given cell can collide only with the objects in the eight other cells surrounding it. Look at Figure 1-7, for example.

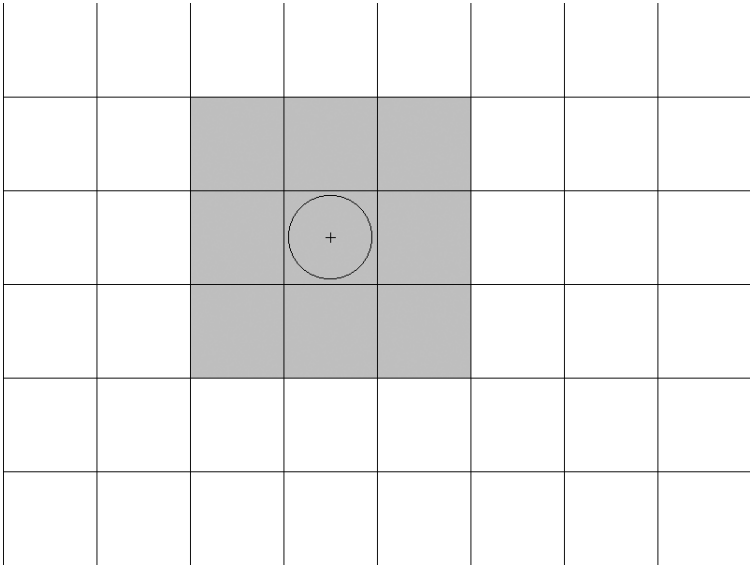


**Figure 1-7.** The ball can collide only with objects in the shaded cells.

The ball shown is assigned to a cell based on its center point. The only objects it can hit are those in the shaded cells. There is no way it can collide with an object in any of the white cells. Even if the ball were on the very edge of that cell, and another ball were on the very edge of a white cell, they could not touch each other (see Figure 1-8).

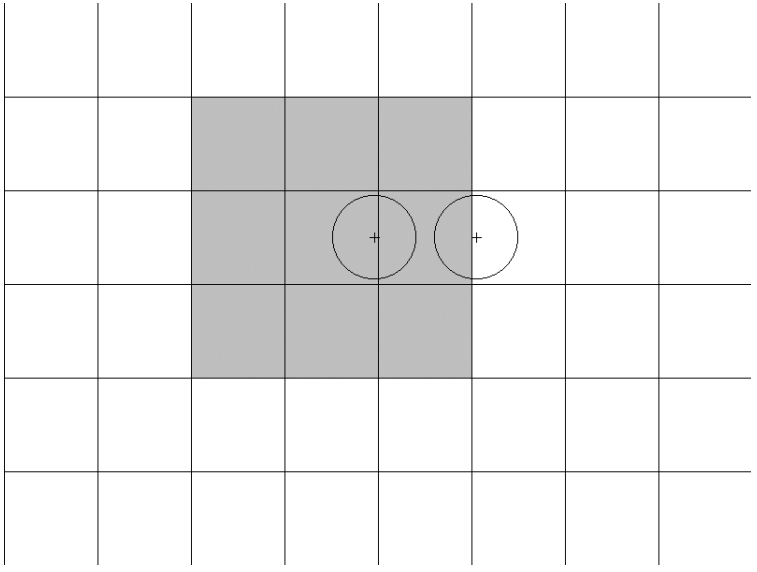


**Figure 1-8.** There's no way the two balls can collide.

Again, this scenario depends on the size of the cells being at least as large as the largest object you will be comparing. If either of the balls were larger than the cells, it would be possible for them to hit each other in the above scenario.

Okay, that's the basic setup. Knowing that, there are probably a number of ways to proceed. I'm not sure there is a single best way, but the goal is to test each object against all the other objects it could possibly reach and make sure that you never test any two objects against each other twice. That's where things get a bit tricky.

I'll outline the method I came up with, which will seem pretty abstract. Just try to get an idea of which areas of the grid we'll be doing collision detection with. Exactly how we'll do all that will be discussed next.

## Implementing grid-based collision detection

We'll start in the upper-left corner. I'll reduce the grid size a bit to make things simpler. See Figure 1-9.

You'll want to test all the objects in that first darker cell with all the objects in all the surrounding cells. Of course, there are no cells to the left or above it, so you just need to check the three light gray cells. Again, there is no way that an object in that dark gray cell can possibly hit anything in any of the white cells.

When that's done, we move on to the next cell. See Figure 1-10.

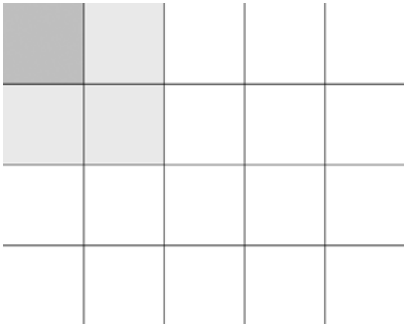


**Figure 1-9.** Test all the objects in the first cell with all the objects in the surrounding cells.

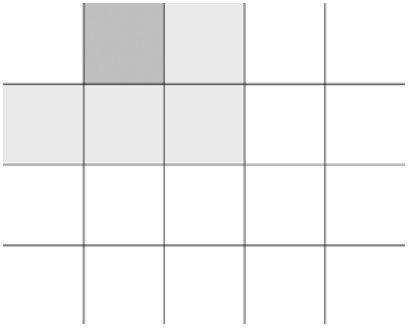


**Figure 1-10.** Continuing with the next cell

With this one, there are a couple more available cells surrounding it, but remember that we already compared all the objects in that first cell with all the objects in the three surrounding cells, which includes the one being tested now. So there is no need to test anything with the first cell again.

We continue across the first row in the same fashion. We only need to test the current cell, the cell to its right, and the three cells below it. See Figures 1-11, 1-12, and 1-13.

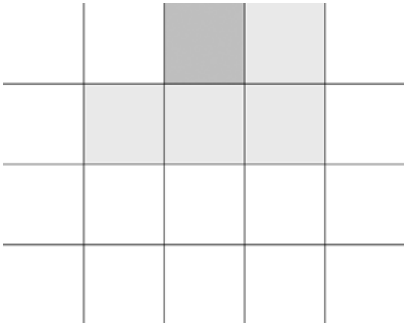


**Figure 1-11.** Continuing across the first row

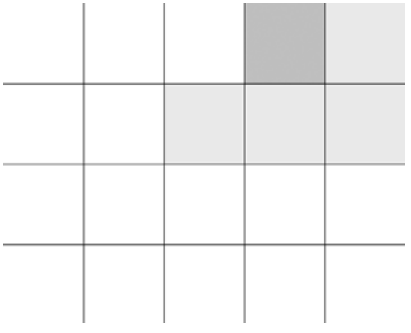


**Figure 1-12.** Next column in first row

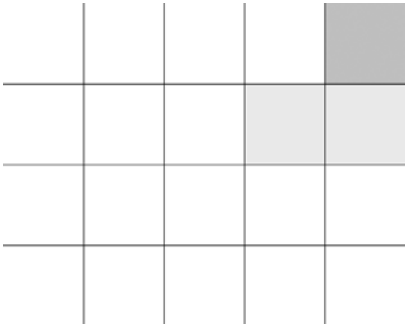


**Figure 1-13.** Final column in first row