

Foundation ActionScript 3.0 Animation

Making Things Move!

Keith Peters



Foundation ActionScript 3.0 Animation: Making Things Move!

Copyright © 2007 by Keith Peters

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN-13 (pbk): 978-1-59059-791-0

ISBN-10 (pbk): 1-59059-791-5

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

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

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 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit www.apress.com.

The information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

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

Credits

Lead Editor **Assistant Production Director**
Chris Mills Kari Brooks-Copony

Technical Reviewer **Compositor**
Todd Yard Dina Quan

Editorial Board **Artist**
Steve Anglin, Ewan Buckingham,
Gary Cornell, Jason Gilmore,
Jonathan Gennick, Jonathan Hassell, April Milne
James Huddleston, Chris Mills, **Proofreader**
Matthew Moodie, Jeff Pepper, April Eddy
Paul Sarknas, Dominic Shakeshaft, **Indexer**
Jim Sumser, Matt Wade John Collin

Project Manager | Production Editor **Cover Image Designer**
Laura Esterman Corné van Dooren

Copy Edit Manager **Interior and Cover Designer**
Nicole Flores Kurt Krames

Copy Editors **Manufacturing Director**
Nicole Flores, Ami Knox Tom Debolski



Chapter 8

EASING AND SPRINGING

What we'll cover in this chapter:

- Proportional motion
- Easing
- Springing
- Important formulas in this chapter

It's hard to believe that it took seven chapters to get through “the basics,” but here you are at Chapter 8, the beginning of the advanced stuff. Or, as I prefer to think of it, the point where things start to get *really* interesting. Up to now, each chapter covered more general techniques and concepts. Beginning with this chapter, I'll be concentrating on one or two specialized types of motion per chapter.

In this chapter, I'll cover easing (proportional velocity) and springing (proportional acceleration). But don't think that because there are only two items, this is a chapter you can skim through quickly. I've gotten more mileage from these two techniques than just about any others. And I'm going to run you through plenty of examples, so you can get an idea of just how powerful these techniques are.

Proportional Motion

Easing and springing are closely related. Even though I had planned from the beginning to put these two subjects in the same chapter, it wasn't until I sat down to plan this chapter in more detail that I realized just *how* similar they are to each other.

Both techniques involve moving an object (usually a sprite or movie clip) from an existing position to a target position. In easing, the sprite kind of slides into the target and stops. In springing, it bounces around back and forth for a bit, and then finally settles down at the target. The two techniques have the following in common:

- You set a target.
- You determine the distance to that target.
- Your motion is proportional to that distance—the bigger the distance, the more the motion.

The difference between easing and springing is in what aspect of the motion is proportional. In easing, *velocity* is proportional to the distance; the further away from the target, the faster the object moves. As it gets very, very close to the object, it's hardly moving at all.

In springing, *acceleration* is proportional to the distance. If the object is far away from the target, a whole lot of acceleration is applied, increasing the velocity quickly. As the object gets closer to its target, less acceleration is applied, but it's still accelerating! It flies right past the target, and then acceleration pulls it back. Eventually, friction causes it to settle down.

Let's dive in to each technique separately, starting with easing.

Easing

One thing I want to clear up right off the bat is that there is more than one type of easing. Even in the Flash IDE, while you're making a motion tween, you have the ability to “ease in” or “ease out.” The type of easing I will be discussing here is the same as the “ease out” of a motion tween. A bit later in this chapter, in the “Advanced easing” section, I'll provide you with a link where you can find out how to do all kinds of easing.

Simple easing

Simple easing is a very basic concept. You have something over here and you want to move it over there. Since you're creating the "illusion of motion," you want to move it there gradually, over several frames. You could simply find the angle between the two, set a speed, use some trigonometry to work out the v_x and v_y , and set it in motion. Then you could check the distance to the target on each frame (using the Pythagorean Theorem, as described in Chapter 3), and when it was there, stop it. That approach might actually be adequate in some situations, but if you're trying to make something look like it's moving naturally, it won't do.

The problem is that your object would move along at a fixed velocity, reach its target, and stop dead. If you're talking about some object moving along and hitting a brick wall, yes, it might be sort of like that. But when you're moving an object to a *target*, this generally implies that someone or something knows where this target is, and is moving something into place there deliberately. In such a case, the motion will start out fairly fast, and then slow down as it gets closer to the target. In other words, its velocity is going to be proportional to the distance to the target.

Let's take an example. You're driving home. When you are a few miles away, you're going to be traveling as fast as the speed limit allows you (OK, maybe even faster—maybe even fast enough to earn yourself a ticket). When you pull off the highway and into your neighborhood, you'll be going a bit slower. Once you're on your own street, a block or two away, you'll be going far slower. As you approach your driveway, you're down to a few miles per hour. When you reach the last few feet of the driveway, you're moving a lot slower than when you pulled into the driveway. And inches before you stop, you're moving at a fraction of that speed.

If you take the time to look, you'll see this behavior manifests itself even in small things like closing a drawer or door. You start out fast and gradually slow down. The next time you go to close a door, make an effort to follow through with the same speed you started with. Just be prepared to explain to anyone nearby why you're slamming doors.

So, when you use easing to move an object into position, it automatically takes on a very natural appearance. One of the coolest things is that simple easing is actually very easy to do. In fact, it's probably easier than figuring out the angle, the v_x , and the v_y , and moving at a fixed speed.

Here is the strategy for easing:

1. Decide on a number for your proportional motion. This will be a fraction of 1.
2. Determine your target.
3. Calculate the distance from the object to the target.
4. Multiply the distance by the fraction. This is your velocity.
5. Add the velocity value to the current position.
6. Repeat steps 3 through 5 until the object is at the target.

Figure 8-1 illustrates the concept.

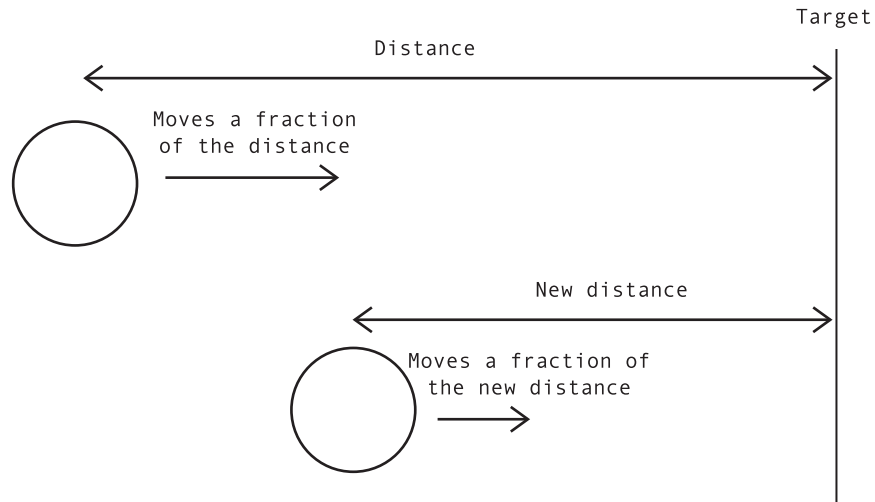


Figure 8-1. Basic easing

Let's go through these steps one at a time, and see how each looks in ActionScript. Don't worry about where to put the code yet. I'm just showing you what the code looks like and what it means.

First, decide on a fraction to represent the proportion. As I said, the velocity will be proportional to the motion. Specifically, this means that the velocity will be a fraction of the distance, something between 0 and 1. The closer it is to 1, the quicker the object will move. The closer it is to 0, the slower it will move, but be careful, because too low a value will prevent the object from reaching the target at all. For starters, choose something like 0.2. I'm going to call this variable *easing*. So you can start off with the following code:

```
var easing:Number = 0.2;
```

Next, determine your target. This is a simple x, y position. You can make it center stage for lack of anything better.

```
var targetX:Number = stage.stageWidth / 2;
var targetY:Number = stage.stageHeight / 2;
```

Then calculate the distance to the target. Assuming you have a sprite named *ball*, you just subtract the ball's x and y from the target x and y.

```
var dx:Number = targetX - ball.x;
var dy:Number = targetY - ball.y;
```

Your velocity is then the distance times the fraction:

```
vx = dx * easing;
vy = dy * easing;
```

And you know what to do from there:

```
ball.x += vx;
ball.y += vy;
```

Steps 3 to 5 need to be repeated, so those will go in your `enterFrame` handler. Let's take a closer look at those three steps, as they can be largely simplified:

```
var dx:Number = targetX - ball.x;
var dy:Number = targetY - ball.y;
vx = dx * easing;
vy = dy * easing;
ball.x += vx;
ball.y += vy;
```

You can condense the first two lines into the second two pretty easily:

```
vx = (targetX - ball.x) * easing;
vy = (targetY - ball.y) * easing;
ball.x += vx;
ball.y += vy;
```

Or, if you're into the whole brevity thing, you can shrink it even further:

```
ball.x += (targetX - ball.x) * easing;
ball.y += (targetY - ball.y) * easing;
```

The first few times you do easing, you might want to go with one of the more verbose syntaxes to make it clearer. But once you've done it a few hundred times, the third version communicates perfectly. I'll stick with the second version here, just to reinforce the idea that you're dealing with velocity.

Now, let's see it in action. You'll need the `Ball` class you've been using all along. Here's the document class, `Easing1.as`:

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

    public class Easing1 extends Sprite
    {
        private var ball:Ball;
        private var easing:Number = 0.2;
        private var targetX:Number = stage.stageWidth / 2;
        private var targetY:Number = stage.stageHeight / 2;

        public function Easing1()
        {
            init();
        }
    }
}
```

```
private function init():void
{
    ball = new Ball();
    addChild(ball);
    addEventListener(Event.ENTER_FRAME, onEnterFrame);
}

private function onEnterFrame(event:Event):void
{
    var vx:Number = (targetX - ball.x) * easing;
    var vy:Number = (targetY - ball.y) * easing;
    ball.x += vx;
    ball.y += vy;
}
}
```

Play around with the easing variable to see how it affects the resulting motion.

The next thing you might want to do is make the ball draggable so you can move it around and see how it always goes back to its target. This is pretty similar to the drag-and-drop technique you set up in Chapter 7. Here, you start dragging the ball when it is pressed with the mouse. At the same time, you remove the enterFrame event handler and listen for mouseUp on the stage. On the mouseUp handler, you stop dragging, remove the mouseUp handler, and reenables the enterFrame. Here's the document class, Easing2.as:

```
package
{
    import flash.display.Sprite;
    import flash.events.Event;
    import flash.events.MouseEvent;

    public class Easing2 extends Sprite
    {
        private var ball:Ball;
        private var easing:Number = 0.2;
        private var targetX:Number = stage.stageWidth / 2;
        private var targetY:Number = stage.stageHeight / 2;

        public function Easing2()
        {
            init();
        }

        private function init():void
        {
            ball = new Ball();
            addChild(ball);
            ball.addEventListener(MouseEvent.MOUSE_DOWN, onMouseDown);
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }
    }
}
```

```

private function onMouseDown(event:MouseEvent):void
{
    ball.startDrag();
    removeEventListener(Event.ENTER_FRAME, onEnterFrame);
    stage.addEventListener(MouseEvent.MOUSE_UP, onMouseUp);
}

private function onMouseUp(event:MouseEvent):void
{
    ball.stopDrag();
    addEventListener(Event.ENTER_FRAME, onEnterFrame);
    stage.removeEventListener(MouseEvent.MOUSE_UP, onMouseUp);
}

private function onEnterFrame(event:Event):void
{
    var vx:Number = (targetX - ball.x) * easing;
    var vy:Number = (targetY - ball.y) * easing;
    ball.x += vx;
    ball.y += vy;
}
}
}

```

When to stop easing

If you are doing simple easing to a single target, eventually you'll get to the point where the object is at the target and the purpose of the easing has been achieved. But, in the examples so far, the easing code continues to execute, even though the object isn't visibly moving anymore. If you are just easing to that point and leaving the object there, continuing to run the easing code is a waste of CPU resources. If you've reached your goal, you might as well stop trying. At first glance, this would be as simple as checking whether the object is at its target and turning off the enterFrame code, like so:

```

if(ball.x == targetX && ball.y == targetY)
{
    // code to stop the easing
}

```

But it winds up being a little more tricky.

The type of easing we are discussing involves something known as *Xeno's Paradox*. Xeno was yet another Greek guy who liked to measure things. Xeno tried to break down motion as follows: In order for something to move from point A to point B, it first must move to a point halfway between the two. Then it needs to travel from that point to a point halfway between there and point B. And then halfway again. Since you always need to move halfway to the target, you can never actually reach the target.

The paradox is that this sounds logical, but obviously we do move from one point to another every day, so there's something wrong there. Let's take a look at it in Flash. On the x axis, a sprite is at

position 0. Say you want to move it to 100 on the x axis. To fit into the paradox, make the easing variable 0.5, so it always moves half the distance to the target. It progresses like this:

- Starting at 0, after frame 1, it will be at 50.
- Frame 2 will bring it to 75.
- Now the distance is 25. Half of that is 12.5, so the new position will be 87.5.
- Following the sequence, the position will be 93.75, 96.875, 98.4375, and so on. After 20 frames, it will be 99.999809265.

As you can see, it gets closer and closer but never actually reaches the target—theoretically. However, things are a bit different when you examine what the code does. It comes down to the question, “How small can you slice a pixel?” The answer is 20. In fact, there’s a name for a twentieth of a pixel: a *twip*. Internally, Flash calculates anything that uses pixels with twips. This includes positions of sprites, movie clips, and any other object on the stage. Thus, if you trace the position of a sprite, you might notice that it is always in multiples of 0.05.

In the example here, the closest a sprite can get to 100 without actually getting there is 99.95. When you try to split the difference at that point, you’re trying to add on $(100 - 99.95) / 2$. This comes out to 0.025, or a fortieth of a pixel. But a twip is as low as you can go. You can’t add “half a twip,” so you wind up adding 0. If you don’t believe me, try the following code (hint: put it in the `init` method of the skeleton class I gave you in Chapter 2):

```
var sprite:Sprite;
sprite = new Sprite();
addChild(sprite);
sprite.x = 0;
var targ:Number = 100;
for(var i:Number=0;i<20;i++)
{
    trace(i + ": " + sprite.x);
    sprite.x += (targ - sprite.x) * .5;
}
```

This just loops through 20 times, moving the sprite half the distance to the target. It’s basic easing code. I threw it in a `for` loop because I was only interested in tracing the positions, not actually seeing the motion. What you’ll find is that by the eleventh iteration, the ball has reached 99.95, and that’s as far as it gets.

In tracing positions, you might also notice that occasionally you get a number like 96.85000000000001. This has to do with the way fractions are stored in a binary format, and has nothing to do with pixels, twips, or Flash itself. For more information, do a web search for “binary round-off errors.”

To make a long story short, no, your sprite object is not going to get closer and closer, but yes, your sprite might never reach its target. So, if you’re doing a simple comparison, as in the previous example, your easing code will never get shut off. What you need to do is answer the question, “When is it close enough?” This comes down to determining whether the distance to the target is less than a certain

amount. For many applications, I've found that if an object is within a pixel of its target, it's safe to say it has arrived, and I can shut off easing.

If you are dealing with two dimensions, you can calculate the distance using the formula I introduced in Chapter 3:

```
distance = Math.sqrt(dx * dx + dy * dy)
```

If you are dealing with a single value for distance, as when you are moving an object on a single axis, you need to use the absolute value of that distance, as it may be negative. You can do this by using the `Math.abs` method.

OK, I've done way too much talking, or writing anyway. Let's see it in some code. Here's a simple document class to demonstrate turning off easing (`EasingOff.as`):

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

    public class EasingOff extends Sprite
    {
        private var ball:Ball;
        private var easing:Number = 0.2;
        private var targetX:Number = stage.stageWidth / 2;

        public function EasingOff()
        {
            init();
        }

        private function init():void
        {
            ball = new Ball();
            addChild(ball);
            ball.y = stage.stageHeight / 2;
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        private function onEnterFrame(event:Event):void
        {
            var dx:Number = targetX - ball.x;
            if(Math.abs(dx) < 1)
            {
                ball.x = targetX;
                removeEventListener(Event.ENTER_FRAME, onEnterFrame);
                trace("done");
            }
            else

```

```
        {
            var vx:Number = dx * easing;
            ball.x += vx;
        }
    }
}
```

As you can see, this example expands the easing formula a bit to first calculate the distance, since you'll need this to see whether easing should be stopped. Perhaps now you can see why you need to use the absolute value of dx. If the ball were to the right of the target, dx would wind up as a negative number, the statement `if(dx < 1)` would evaluate as true, and that would be the end of things. By using `Math.abs`, you make sure that the actual distance is less than 1. You then place the ball where it is trying to go and disable the motion code.

Remember that if you are doing something like a drag-and-drop with easing, you'll want to reenable the motion code when the ball is dropped. Why don't you go ahead and see whether you can figure that one out for yourself.

A moving target

In the examples so far, the target point has been a single, fixed location, but that's not a requirement. The distance is calculated on each frame, and the velocity is then calculated based on that. Flash doesn't care whether or not it reaches the target, or if the target keeps moving. It just happily goes on saying, "Where's my target? What's the distance? What's my velocity?" on each and every frame.

You can easily make the mouse an easing target. Just plug in the mouse coordinates (`mouseX` and `mouseY`) where you had `targetX` and `targetY` before. Here's a simpler version that does just that (`EaseToMouse.as`):

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

    public class EaseToMouse extends Sprite
    {
        private var ball:Ball;
        private var easing:Number = 0.2;

        public function EaseToMouse()
        {
            init();
        }

        private function init():void
        {
            ball = new Ball();
        }
    }
}
```

```

        addChild(ball);
        addEventListener(Event.ENTER_FRAME, onEnterFrame);
    }

    private function onEnterFrame(event:Event):void
    {
        var vx:Number = (mouseX - ball.x) * easing;
        var vy:Number = (mouseY - ball.y) * easing;
        ball.x += vx;
        ball.y += vy;
    }
}

```

Move the mouse around and see how the ball follows, and how it goes faster when you get further away.

Think of what other moving targets you could have. Maybe a sprite could ease to another sprite. Back in the early days of Flash, mouse trailers—a trail of movie clips that followed the mouse around—were all the rage. Easing was one way of doing this. The first clip eased to the mouse, the second clip eased to the first, the third to the second, and so on. Go ahead and try it out (but if you put it on a live website, I'll deny any responsibility).

Easing isn't just for motion

If there is one point I want to get across in this book, it's that the examples I give are simply that: examples. In each case, I'm mainly just manipulating numbers that are used for various properties of a sprite or other object. For the most part, I'm using the x and y properties to control positions of sprites. Just remember that sprites, movie clips, and other display objects have many other properties that you can manipulate, and most of them are represented by numbers. So when you read a particular example, definitely try it out, but don't leave it at that. Try the same example on other properties. Here, I'll give you a few ideas to get you started.

Transparency

Apply easing to the alpha property of a clip. Start out by setting it to 0, and making the target 1 (realizing that in AS 3, alpha is a value from 0.0 to 1.0):

```

ball.alpha = 0;
var targetAlpha:Number = 1;

```

Then you can fade it in with easing in an enterFrame handler:

```

ball.alpha += (targetAlpha - ball.alpha) * easing;

```

Or reverse the 0 and 1 to make it fade out.

Rotation

Set the rotation property and a target rotation. Of course, you need something that can be visibly rotated, like an arrow:

```
arrow.rotation = 90;  
var targetRotation:Number = 270;
```

And then ease it:

```
arrow.rotation += (targetRotation - arrow.rotation) * easing;
```

Colors

If you are up for a real challenge, try easing on 24-bit colors. You'll need to start with red, green, and blue initial values and target values, perform the easing on each separate component color, and then combine them into a 24-bit color. For instance, you could ease from red to blue. Start with the initial and target colors:

```
red = 255;  
green = 0;  
blue = 0;  
redTarget = 0;  
greenTarget = 0;  
blueTarget = 255;
```

Then in your enterFrame handler, perform easing on each one. Here is just the red value:

```
red += (redTarget - red) * easing;
```

Then combine the three into a single value (as described in Chapter 4):

```
col = red << 16 | green << 8 | blue;
```

And use that in ColorTransform (see Chapter 4) or for a line color, a fill color, or anyplace else where color is used.

Advanced easing

Now that you've seen how simple easing works, you might consider using more complex easing formulas for additional effects. For instance, you might want something to start slowly, build up speed, and then slow down as it approaches its target. Or you might want to ease something into position over a certain time period or number of frames.

Robert Penner has become famous for collecting easing formulas, cataloging them, and implementing them in ActionScript. You can find his easing formulas at www.robertpenner.com. At the time of writing this, there were no AS 3 versions, but they would be easy enough to convert to AS 3, knowing just what you have learned so far in this book.

OK, let's move on to perhaps my most favorite subject in Flash: springing.

Springing

Maybe it's just me, but I've always found springing to be one of the most powerful and useful physics concepts in ActionScripted animation. It seems like you can do almost anything with a spring. Of course, it's just another technique that has its specific uses. I've gotten so much mileage out of this one, however, that I get a bit enthusiastic about it. So, let's look at what a spring is and how you can program it in Flash.

As I mentioned at the beginning of the chapter, a spring's acceleration is proportional to its distance from a target. Think about a real, physical spring, or better yet, a ball on the end of a rubber band. You attach the other end to something solid. As the ball is hanging there with no force applied, that's its target point. That's where it wants to be. Now pull it away a tiny bit and let it go. At that point, the ball's velocity is zero, but the rubber band applies force to it, pulling it back to the target. Now pull it away as far as you can and let it go. The rubber band applies a lot more force. The ball zooms right past its target and starts going the other way. Its velocity is very high. But when it gets a little bit past the target, the rubber band starts pulling it back a bit—changes its velocity. The ball keeps going, but the farther it goes, the more the band pulls back on it. Eventually, the velocity reaches zero, the direction reverses, and the whole thing starts over again. Finally, after bouncing back and forth a few times, it slows down and comes to a stop at—you guessed where—its target.

Now, let's start translating this into ActionScript so you can use it. To keep things simple, let's start off with one dimension.

Springing in one dimension

Let's drag our good friend, the red ball, back into active service. You'll leave it over at its default x position of zero and have it spring to the middle. As with easing, you'll need a variable to hold the proportionate value of the spring. You can think of this as the proportion of the distance that will be added to the velocity. A high spring value will make a very stiff spring. Something lower will look more like a loose rubber band. You'll start off with 0.1. Here's what you have so far:

```
private var spring:Number = 0.1;
private var targetX:Number = stage.stageWidth / 2;
private var vx:Number = 0;
```

Again, don't worry about where to put this just yet. Just make sure you know what these variables and statements are about.

Then move to the motion code and find the distance to the target:

```
var dx:Number = targetX - ball.x;
```

Now, compute some acceleration. The acceleration will be proportional to that distance. In fact, it will be the distance multiplied by the spring value:

```
var ax:Number = dx * spring;
```

Once you have a value for acceleration, you should be back on familiar ground. Add the acceleration to the velocity and add the velocity to the position, right?

```
vx += ax;
ball.x += vx;
```

Before you write any code, let's simulate it with some sample numbers. Let's say the x position is 0, vx is 0, the target x is 100, and the spring variable is 0.1. Here is how it might progress:

1. Multiply distance (100) by spring, and you get 10. Add that to vx, which then becomes 10. Add velocity to position, making the x position 10.
2. Next round, distance (100 – 10) is 90. Acceleration is 90 times 0.1, or 9 this time. This gets added to vx, which becomes 19. The x position becomes 29.
3. Next round, distance is 71, acceleration is 7.1, which added to vx makes it 26.1. The x position becomes 55.1.
4. Next round, distance is 44.9, acceleration is 4.49, and vx becomes 30.59. The x position is then 85.69.

The thing to note is that the acceleration on each frame becomes less and less as the object approaches its target, but the velocity continues to build. It's not building as rapidly as it was on previous frames, but it's still moving faster and faster.

After a couple more rounds, the object goes right past the target to an x position of around 117. The distance is now 100 – 117, which is –17. A fraction of this gets added to the velocity, slowing the object down a bit.

Now that you understand how springing works, let's make a real file. As usual, make sure the Ball class is available, and use the following document class (Spring1.as):

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

    public class Spring1 extends Sprite
    {
        private var ball:Ball;
        private var spring:Number = 0.1;
        private var targetX:Number = stage.stageWidth / 2;
        private var vx:Number = 0;

        public function Spring1()
        {
            init();
        }

        private function init():void
        {
            ball = new Ball();
            addChild(ball);
            ball.y = stage.stageHeight / 2;
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }
    }
}
```

```

private function onEnterFrame(event:Event):void
{
    var dx:Number = targetX - ball.x;
    var ax:Number = dx * spring;
    vx += ax;
    ball.x += vx;
}
}
}

```

Test this, and you'll see you definitely have something spring-like going on there. The only problem is that it kind of goes on forever. Earlier, when I described a spring, I said that “it slows down and comes to a stop.” As is though, the ball builds up the same velocity on each leg of its swing, so it keeps bouncing back and forth at the same speed. You need something to reduce its velocity and slow it down. Hmmm . . . sound familiar? That's right, you need to apply some friction. Easy enough—just make a friction variable, with a value like 0.95 for starters. This goes up at the top of the class with the rest of the class variables:

```
private var friction:Number = 0.95;
```

Then multiply vx by friction somewhere in the enterFrame handler. Here's the corrected onEnterFrame method in document class Spring2.as:

```

private function onEnterFrame(event:Event):void
{
    var dx:Number = targetX - ball.x;
    var ax:Number = dx * spring;
    vx += ax;
    vx *= friction;
    ball.x += vx;
}

```

At this point, you have a full-fledged, albeit one-dimensional, spring. Definitely play with this one a lot. See what different values for spring and friction do and how they interact. Check out how a different starting position or target position affects the action of the system, the speed of the ball, and the rate at which it slows down and comes to a stop. Knowing this one file well will take you a long way. When you've got it down cold, you're ready to move on to a two-dimensional spring.

Springing in two dimensions

If you are thinking that a two-dimensional spring is as simple as adding a second target, velocity, and acceleration, I have news for you: You're right. So, without further ado, here is a two-dimensional spring (Spring3.as):

```

package
{
    import flash.display.Sprite;
    import flash.events.Event;

```

```
public class Spring3 extends Sprite
{
    private var ball:Ball;
    private var spring:Number = 0.1;
    private var targetX:Number = stage.stageWidth / 2;
    private var targetY:Number = stage.stageHeight / 2;
    private var vx:Number = 0;
    private var vy:Number = 0;
    private var friction:Number = 0.95;

    public function Spring3()
    {
        init();
    }

    private function init():void
    {
        ball = new Ball();
        addChild(ball);
        addEventListener(Event.ENTER_FRAME, onEnterFrame);
    }

    private function onEnterFrame(event:Event):void
    {
        var dx:Number = targetX - ball.x;
        var dy:Number = targetY - ball.y;
        var ax:Number = dx * spring;
        var ay:Number = dy * spring;
        vx += ax;
        vy += ay;
        vx *= friction;
        vy *= friction;
        ball.x += vx;
        ball.y += vy;
    }
}
```

As you can see, the only difference is adding in all the y-axis stuff. The problem is, it still seems rather one-dimensional. Yes, the ball is now moving on the x and y axes, but it's just going in a straight line. That's because its velocity starts out as zero, and the only force acting on it is the pull toward the target, so it goes in a straight line toward its target.

To make things a little more interesting, initialize vx to something other than 0. Try something nice and big like 50. Now, you have something that looks a little more loose and fluid. But you're only getting started. It gets a lot cooler.

Springing to a moving target

You probably won't be surprised to hear that springing doesn't require the target to be the same on each frame. When I covered easing, I gave you a quick and easy example of the ball following the mouse. It's pretty easy to adapt that example to make the ball spring to the mouse. Instead of the `targetX` and `targetY` you've been using, use the mouse coordinates. In springing, as with easing, the distance to the target is always calculated newly on each frame. Acceleration is based on that, and that acceleration is added to the velocity.

The effect is so cool, I feel like I should be writing on and on about it. But the fact is, there's really not much more to say on the subject, and the code isn't all that different. In the preceding example, simply change these lines:

```
var dx:Number = targetX - ball.x;
var dy:Number = targetY - ball.y;
```

to read like this:

```
var dx:Number = mouseX - ball.x;
var dy:Number = mouseY - ball.y;
```

You can also remove the lines that declare the `targetX` and `targetY` variables, though they are not going to hurt anything if you leave them in. The updated document class is available as `Spring4.as`.

This is another good point to stop and play. Get a really good feeling for how all these variables work, and try out many variations. Break it. Find out what breaks it. Have fun with it!

So where's the spring?

At this point, you have a very realistic-looking ball on the end of a rubber band. But it seems to be an invisible rubber band. Well, you can remedy that pretty easily with a few lines of drawing API code!

Since you have a fairly simple file without much else going on, you can safely apply your drawing code directly to the main class, which extends the `Sprite` class. In a more complex application, you might want to create an empty sprite and use that as a kind of drawing layer.

The strategy is simple. In each frame, after the ball is in position, you call `clear()` to erase any previous lines. Then you reset `lineStyle` and draw a line from the ball to the mouse. You just need the following code in the `enterFrame` handler, immediately after you set the ball's position (you'll see it in the full code shortly):

```
graphics.clear();
graphics.lineStyle(1);
graphics.moveTo(ball.x, ball.y);
graphics.lineTo(mouseX, mouseY);
```

Well, this is fun! What else can you do? How about adding some gravity so the ball looks like it's actually hanging off the end of the mouse? That's easy. Just add a gravity variable and add that to the `vy`

for each frame. You know the drill by now. The following code (Spring5.as) incorporates the line drawing and gravity additions:

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

    public class Spring5 extends Sprite
    {
        private var ball:Ball;
        private var spring:Number = 0.1;
        private var vx:Number = 0;
        private var vy:Number = 0;
        private var friction:Number = 0.95;
        private var gravity:Number = 5;

        public function Spring5()
        {
            init();
        }

        private function init():void
        {
            ball = new Ball();
            addChild(ball);
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        private function onEnterFrame(event:Event):void
        {
            var dx:Number = mouseX - ball.x;
            var dy:Number = mouseY - ball.y;
            var ax:Number = dx * spring;
            var ay:Number = dy * spring;
            vx += ax;
            vy += ay;
            vy += gravity;
            vx *= friction;
            vy *= friction;
            ball.x += vx;
            ball.y += vy;
            graphics.clear();
            graphics.lineStyle(1);
            graphics.moveTo(ball.x, ball.y);
            graphics.lineTo(mouseX, mouseY);
        }
    }
}
```

When you test this version, you should see something like Figure 8-2.

Notice how you need to increase the gravity variable to 5 in order to get the ball to actually hang down. Much less than that, and the force of the spring overcomes the force of gravity and you don't see the effect.

Now, here's another point where I've just butchered real-world physics. Of course, you can't go around "increasing gravity" on objects! Gravity is a constant, based on the size and mass of the planet you happen to be on. What you can do is increase the mass of the object, so that gravity has more of an effect on it. So, technically I should keep gravity at something like 0.5, and then create a mass property and make it something like 10. Then I could multiply mass by gravity and come up with 5 again. Or I could change the name of the gravity variable to something like `forceThatGravityIsExertingOnThisObjectBasedOnItsMass`. But as long as you know that's what I mean, I'll save the space and shorten it to gravity.

Again, experiment with this one. Try decreasing the gravity and spring values. Try changing the friction value. You'll see you can have a nearly endless number of combinations, allowing you to create all kinds of systems.

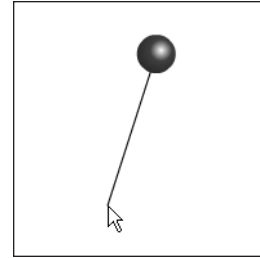


Figure 8-2.
Springing from the mouse,
with a visible spring

Chaining springs

Moving right along, let's chain a few springs together. In the easing section, I discussed mouse trailers briefly, where one object eases to the mouse, another object eases to that object, and so on. I didn't give you an example because it's an old, and somewhat cheesy, effect. But, when you apply the same concept using springing instead, well, that's just different.

Here's the plan: Start off by creating three balls, named `ball0`, `ball1`, and `ball2`. The first one, `ball0`, will behave pretty much like the single ball did in the previous example. Then `ball1` will spring to `ball0`, and `ball2` will spring to `ball1`. All will have gravity, so they should kind of hang down in a chain. The code isn't really anything you haven't seen before, just a little more complex. Here it is, in document class `Chain.as`:

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

    public class Chain extends Sprite
    {
        private var ball0:Ball;
        private var ball1:Ball;
        private var ball2:Ball;
        private var spring:Number = 0.1;
        private var friction:Number = 0.8;
        private var gravity:Number = 5;
    }
}
```

```
public function Chain()
{
    init();
}

private function init():void
{
    ball0 = new Ball(20);
    addChild(ball0);
    ball1 = new Ball(20);
    addChild(ball1);
    ball2 = new Ball(20);
    addChild(ball2);
    addEventListener(Event.ENTER_FRAME, onEnterFrame);
}

private function onEnterFrame(event:Event):void
{
    moveBall(ball0, mouseX, mouseY);
    moveBall(ball1, ball0.x, ball0.y);
    moveBall(ball2, ball1.x, ball1.y);

    graphics.clear();
    graphics.lineStyle(1);
    graphics.moveTo(mouseX, mouseY);
    graphics.lineTo(ball0.x, ball0.y);
    graphics.lineTo(ball1.x, ball1.y);
    graphics.lineTo(ball2.x, ball2.y);
}

private function moveBall(ball:Ball,
                           targetX:Number,
                           targetY:Number):void
{
    ball.vx += (targetX - ball.x) * spring;
    ball.vy += (targetY - ball.y) * spring;
    ball.vy += gravity;
    ball.vx *= friction;
    ball.vy *= friction;
    ball.x += ball.vx;
    ball.y += ball.vy;
}
}
```

If you take another look at the `Ball` class, you'll see that each instance of the class gets its own `vx` and `vy` properties, and these are initialized to 0. So, in the `init` method, you just need to create each ball and add it to the display list.

Then in the `onEnterFrame` function, you perform all the springing. Note that rather than duplicating the same code three times for each ball, we include a `moveBall` method, which takes care of all the motion code. This takes a reference to a ball and an `x` and `y` target. You call the function for each ball, passing in the `x` and `y` mouse position for the first ball, and the location of the first and second balls as targets for the second and third.

Finally, when all of the balls are in place, you set a line style, move the drawing cursor to the mouse position, and then draw a line to each successive ball, creating the rubber band holding them all together. Note that the friction here was pushed to 0.8 to force things to settle down a bit quicker.

You could make this class a bit more flexible, by creating an array to hold references to each object in the chain, and looping through that array to move each one, and then draw the lines. This would just take a few changes. First, you'd need a couple new variables for the array and the number of objects to create:

```
private var balls:Array;
private var numBalls:Number = 5;
```

The `init` function creates each object in a `for` loop, and adds it to the array:

```
private function init():void
{
    balls = new Array();
    for(var i:uint = 0; i < numBalls; i++)
    {
        var ball:Ball = new Ball(20);
        addChild(ball);
        balls.push(ball);
    }
    addEventListener(Event.ENTER_FRAME, onEnterFrame);
}
```

Finally, the `onEnterFrame` method has the biggest changes. It starts by setting the line style, moving the drawing position to the mouse position, moving the first ball, and drawing a line to it. Then it loops through all the rest of the balls in the array, moving and drawing a line to each one in turn. You can add as many objects as you want simply by changing the value of the one variable, `numBalls`.

```
private function onEnterFrame(event:Event):void
{
    graphics.clear();
    graphics.lineStyle(1);
    graphics.moveTo(mouseX, mouseY);
    moveBall(balls[0], mouseX, mouseY);
    graphics.lineTo(balls[0].x, balls[0].y);

    for(var i:uint = 1; i < numBalls; i++)
    {
        var ballA:Ball = balls[i-1];
        var ballB:Ball = balls[i];
        moveBall(ballB, ballA.x, ballA.y);
    }
}
```

```

        graphics.lineTo(ballB.x, ballB.y);
    }
}

```

You can see the result in Figure 8-3 and can find this class in `ChainArray.as`.

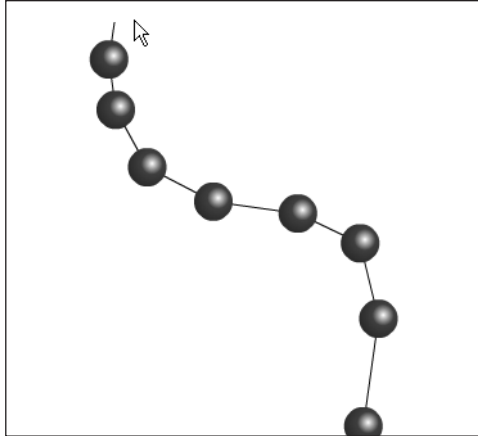


Figure 8-3. Chained springs

Springing to multiple targets

Back when I introduced the subjects of velocity and acceleration in Chapter 5, I talked about how you could have multiple forces acting on an object. Each force is an acceleration, and you just add them on to the velocity, one by one. Well, since a spring is nothing more than something exerting acceleration on an object, it's pretty simple to create multiple springs acting on a single object.

Here's the setup for demonstrating springing to multiple targets: You'll create three "handles," which will just be instances of the `Ball` class, and give them simple drag-and-drop functionality. These will also be targets for the ball to spring to. The ball will try to spring to all three of them at once and find its equilibrium somewhere between them. Or, to put it another way, each target will exert a certain amount of acceleration on the ball, and its motion will be the sum total of all of those forces.

This example gets pretty complex, with several methods in place to handle the various behaviors. I'll give you the code (document class `MultiSpring.as`) and then discuss it section by section.

```

package
{
    import flash.display.Sprite;
    import flash.events.Event;
    import flash.events.MouseEvent;

    public class MultiSpring extends Sprite
    {
        private var ball:Ball;
        private var handles:Array;
    }
}

```

```
private var spring:Number = 0.1;
private var friction:Number = 0.8;
private var numHandles:Number = 3;

public function MultiSpring()
{
    init();
}

private function init():void
{
    ball = new Ball(20);
    addChild(ball);

    handles = new Array();
    for(var i:uint = 0; i < numHandles; i++)
    {
        var handle:Ball = new Ball(10, 0x0000ff);
        handle.x = Math.random() * stage.stageWidth;
        handle.y = Math.random() * stage.stageHeight;
        handle.addEventListener(MouseEvent.MOUSE_DOWN,
            onPress);

        addChild(handle);
        handles.push(handle);
    }

    addEventListener(Event.ENTER_FRAME, onEnterFrame);
    addEventListener(MouseEvent.MOUSE_UP, onRelease);
}

private function onEnterFrame(event:Event):void
{
    for(var i:uint = 0; i < numHandles; i++)
    {
        var handle:Ball = handles[i] as Ball;
        var dx:Number = handle.x - ball.x;
        var dy:Number = handle.y - ball.y;
        ball.vx += dx * spring;
        ball.vy += dy * spring;
    }

    ball.vx *= friction;
    ball.vy *= friction;
    ball.x += ball.vx;
    ball.y += ball.vy;

    graphics.clear();
    graphics.lineStyle(1);
    for(i = 0; i < numHandles; i++)
```

```
        {
            graphics.moveTo(ball.x, ball.y);
            graphics.lineTo(handles[i].x, handles[i].y);
        }
    }

    private function onPress(event:MouseEvent):void
    {
        event.target.startDrag();
    }

    private function onRelease(event:MouseEvent):void
    {
        stopDrag();
    }
}
}
```

The `init` method creates a ball and three handles via a `for` loop, randomly positioning them, and setting them up for drag-and-drop behavior.

The `onEnterFrame` method loops through each handle, springing the ball toward it. It then applies the ball's new velocity to its position, and loops through again, drawing a line from the ball to each handle.

The `onPress` method is pretty straightforward, but note that in the `onRelease` function, you have no way of knowing which handle has been being dragged. Fortunately, calling `stopDrag` on any display object stops all dragging, so you can just call it in the main class.

Note that you can easily change the number of handles to use by simply changing the `numHandles` variable.

Figure 8-4 shows an example of the results of this code.

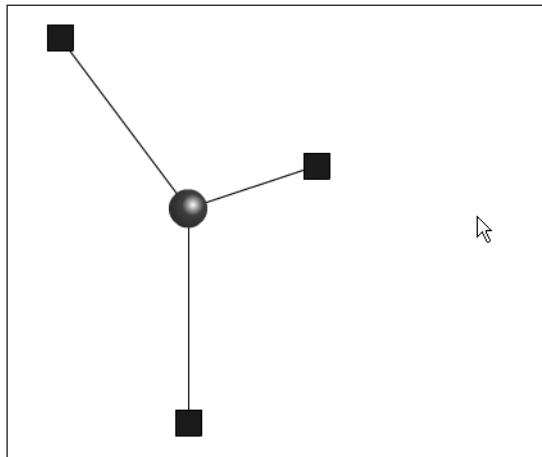


Figure 8-4. Multiple springs

By now, I'm sure you've already started taking many detours and creating a few things that I haven't even mentioned (or perhaps thought of). If so, that's excellent! That's exactly my goal in writing this book.

Offsetting the target

If you took a real spring—an actual coil of bouncy metal—and attached one end of it to something solid and the other end to a ball or some other object, what would be the target? Would the target be the point where the spring is attached? No, not really. The ball would never be able to reach that point, because the spring itself would be in the way. Furthermore, once the spring had contracted to its normal length, it wouldn't be applying any more force on the ball. So, the target would actually be the position of the loose end of the spring in its unstretched state. But that point could vary as the spring pivots around the fixed point.

To find the actual target, you need to first find the angle between the object and the fixed point, and then move out from the fixed point at that angle—the length of the spring. In other words, if the length of the spring were 50, and the angle between the ball and fixed point were 45, you would move out 50 pixels from the fixed point, at an angle of 45 degrees, and that would be the ball's target to spring to. Figure 8-5 illustrates how this works.

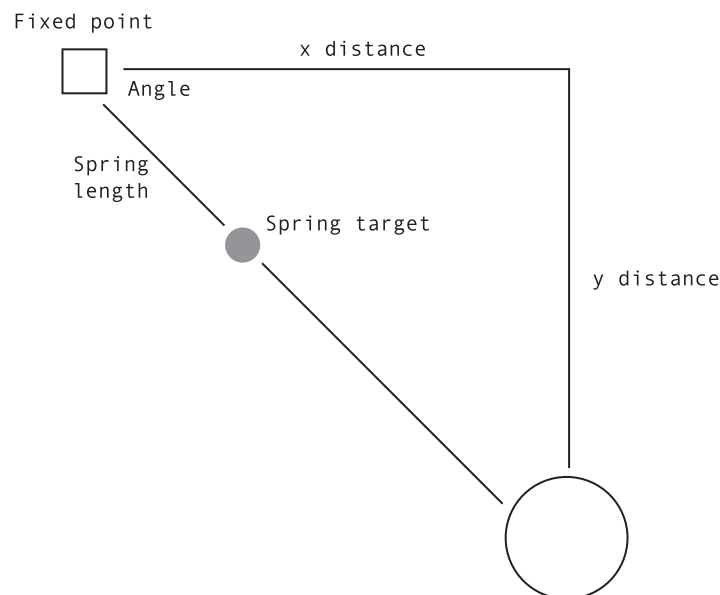


Figure 8-5. Offsetting a spring

The code to find the target in this case is roughly as follows:

```
var dx:Number = ball.x - fixedX;
var dy:Number = ball.y - fixedY;
var angle:Number = Math.atan2(dy, dx);
var targetX:Number = fixedX + Math.cos(angle) * springLength;
var targetY:Number = fixedY + Math.sin(angle) * springLength;
```

So, the result is that the object will spring *toward* the fixed point, but will come to rest some distance away from it. Also, note that although I'm calling it a "fixed point," this just means the point to which the spring is fixed. It doesn't mean that point cannot move. Perhaps it's better just to see it in action. You'll go back to using the mouse position, but this time, it will be the spring's fixed point. The spring's length will be 100 pixels. Here's the document class (OffsetSpring.as):

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

    public class OffsetSpring extends Sprite
    {
        private var ball:Ball;
        private var spring:Number = 0.1;
        private var vx:Number = 0;
        private var vy:Number = 0;
        private var friction:Number = 0.95;
        private var springLength:Number = 100;

        public function OffsetSpring()
        {
            init();
        }

        private function init():void
        {
            ball = new Ball(20);
            addChild(ball);
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        private function onEnterFrame(event:Event):void
        {
            var dx:Number = ball.x - mouseX;
            var dy:Number = ball.y - mouseY;
            var angle:Number = Math.atan2(dy, dx);
            var targetX:Number = mouseX + Math.cos(angle) *
                springLength;
            var targetY:Number = mouseY + Math.sin(angle) *
                springLength;
            vx += (targetX - ball.x) * spring;
            vy += (targetY - ball.y) * spring;
            vx *= friction;
            vy *= friction;
            ball.x += vx;
            ball.y += vy;
            graphics.clear();
        }
    }
}
```

```

        graphics.lineStyle(1);
        graphics.moveTo(ball.x, ball.y);
        graphics.lineTo(mouseX, mouseY);
    }
}
}

```

Even though you can see what is happening here, it might not be too obvious exactly where you would find this technique particularly useful. Well, the next section will give you a specific example.

Attaching multiple objects with springs

I remember the exact moment I realized I could do this. It was something like, “OK, I know how to spring an object to a point. And I know that point does not have to be fixed. I can even have one object spring to another object. Well, what if I have the other object spring back to the first object? So these two objects were linked to each other by a spring. Move either one, and the other one springs to it.”

My initial impression was that it would probably cause some weird feedback loop that would crash Flash, or at least bring up some kind of warning message. But I bravely went ahead and tried it anyway. And to my complete amazement, it worked perfectly!

I’ve already pretty much described the strategy, but to recap: Object A has object B as its target. It springs toward it. Object B in turn has object A as its target. Actually, this is the point where the offset has a great role. If each object had the other as a direct target, they would collapse in on each other and occupy the same point. By applying an offset, you keep them apart a bit, as shown in Figure 8-6.

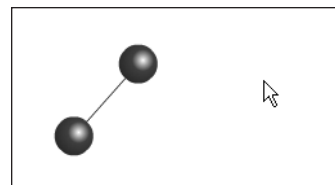


Figure 8-6. Two objects connected by a spring

For this next example, you’ll need two instances of the Ball class. I’ll call them ball0 and ball1. ball0 springs to ball1 with an offset. And ball1 springs to ball0 with an offset. Rather than writing out all the offset, spring, and motion code twice, I put it all into a function called `springTo`. So, you can spring ball0 to ball1 by saying `springTo(ball0, ball1)`, and then spring ball1 to ball0 by saying `springTo(ball1, ball0)`. I also put in a couple of variables, `ball0Dragging` and `ball1Dragging`, to disable the springing for each ball when it is being dragged. Here’s the class (`DoubleSpring.as`):

```

package
{
    import flash.display.Sprite;
    import flash.events.Event;
    import flash.events.MouseEvent;

    public class DoubleSpring extends Sprite
    {
        private var ball0:Ball;
        private var ball1:Ball;
        private var ball0Dragging:Boolean = false;
        private var ball1Dragging:Boolean = false;
    }
}

```

```
private var spring:Number = 0.1;
private var friction:Number = 0.95;
private var springLength:Number = 100;

public function DoubleSpring()
{
    init();
}

private function init():void
{
    ballo = new Ball(20);
    ballo.x = Math.random() * stage.stageWidth;
    ballo.y = Math.random() * stage.stageHeight;
    ballo.addEventListener(MouseEvent.CLICK, onPress);
    addChild(ballo);

    ball1 = new Ball(20);
    ball1.x = Math.random() * stage.stageWidth;
    ball1.y = Math.random() * stage.stageHeight;
    ball1.addEventListener(MouseEvent.CLICK, onPress);
    addChild(ball1);

    addEventListener(Event.ENTER_FRAME, onEnterFrame);
    stage.addEventListener(MouseEvent.CLICK, onRelease);
}

private function onEnterFrame(event:Event):void
{
    if(!balloDragging)
    {
        springTo(ballo, ball1);
    }
    if(!ball1Dragging)
    {
        springTo(ball1, ballo);
    }
    graphics.clear();
    graphics.lineStyle(1);
    graphics.moveTo(ballo.x, ballo.y);
    graphics.lineTo(ball1.x, ball1.y);
}

private function springTo(ballA:Ball, ballB:Ball):void
{
    var dx:Number = ballB.x - ballA.x;
    var dy:Number = ballB.y - ballA.y;
```

```

        var angle:Number = Math.atan2(dy, dx);
        var targetX:Number = ballB.x - Math.cos(angle) *
            springLength;
        var targetY:Number = ballB.y - Math.sin(angle) *
            springLength;
        ballA.vx += (targetX - ballA.x) * spring;
        ballA.vy += (targetY - ballA.y) * spring;
        ballA.vx *= friction;
        ballA.vy *= friction;
        ballA.x += ballA.vx;
        ballA.y += ballA.vy;
    }

    private function onPress(event:MouseEvent):void
    {
        event.target.startDrag();
        if(event.target == ball0)
        {
            ball0Dragging = true;
        }
        if(event.target == ball1)
        {
            ball1Dragging = true;
        }
    }

    private function onRelease(event:MouseEvent):void
    {
        ball0.stopDrag();
        ball1.stopDrag();
        ball0Dragging = false;
        ball1Dragging = false;
    }
}

```

For this file, the balls are placed on stage and are set up for drag-and-drop. The `enterFrame` handler function simply calls the `springTo` function on each ball. Note that in the program, each of these lines is surrounded by a check to make sure the ball isn't being dragged:

```

springTo(ball0, ball1);
springTo(ball1, ball0);

```

The `springTo` function is where all the action happens. Everything in this function should be familiar to you. First, it finds the distance and angle to the other ball, and calculates a target point based on that. It then performs basic spring mechanics on that target point. When the function is called again, with the parameters reversed, the balls swap roles, and the original target ball springs toward the other one. This may not be the most efficient code, but it demonstrates what is happening as clearly as possible.

You'll see that neither ball is attached to any fixed point or the mouse; they are both free-floating. Their only constraint is that they maintain a certain distance from each other. The great thing about this setup is that it is now very easy to incorporate additional objects. For example, if you create a third ball (ball2) and set it up like the others (along with a ball2Dragging variable), you can add that into the mix like so:

```

if(!ball0Dragging)
{
    springTo(ball0, ball1);
    springTo(ball0, ball2);
}
if(!ball1Dragging)
{
    springTo(ball1, ball0);
    springTo(ball1, ball2);
}
if(!ball2Dragging)
{
    springTo(ball2, ball0);
    springTo(ball2, ball1);
}

```

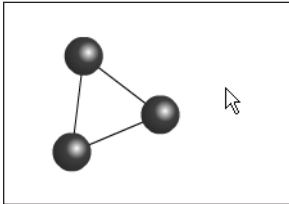


Figure 8-7. Three objects connected by a spring

This will create a triangle formation, as shown in Figure 8-7, which to me is pretty cool. I'm sure once you get the hang of this, you'll quickly move on to a square, and from there, all kinds of complex springy structures.

Important formulas in this chapter

Once again, it's time to review the important formulas presented in this chapter.

Simple easing, long form:

```

var dx:Number = targetX - sprite.x;
var dy:Number = targetY - sprite.y;
vx = dx * easing;
vy = dy * easing;
sprite.x += vx;
sprite.y += vy;

```

Simple easing, abbreviated form:

```
vx = (targetX - sprite.x) * easing;
vy = (targetY - sprite.y) * easing;
sprite.x += vx;
sprite.y += vy;
```

Simple easing, short form:

```
sprite.x += (targetX - sprite.x) * easing;
sprite.y += (targetY - sprite.y) * easing;
```

Simple spring, long form:

```
var ax:Number = (targetX - sprite.x) * spring;
var ay:Number = (targetY - sprite.y) * spring;
vx += ax;
vy += ay;
vx *= friction;
vy *= friction;
sprite.x += vx;
sprite.y += vy;
```

Simple spring, abbreviated form:

```
vx += (targetX - sprite.x) * spring;
vy += (targetY - sprite.y) * spring;
vx *= friction;
vy *= friction;
sprite.x += vx;
sprite.y += vy;
```

Simple spring, short form:

```
vx += (targetX - sprite.x) * spring;
vy += (targetY - sprite.y) * spring;
sprite.x += (vx *= friction);
sprite.y += (vy *= friction);
```

Offset spring:

```
var dx:Number = sprite.x - fixedX;
var dy:Number = sprite.y - fixedY;
var angle:Number = Math.atan2(dy, dx);
var targetX:Number = fixedX + Math.cos(angle) * springLength;
var targetY:Number = fixedY + Math.sin(angle) * springLength;
// spring to targetX, targetY as above
```

Summary

This chapter covered the two basic techniques of proportional motion: easing and springing. You've learned that easing is proportional motion and springing is proportional velocity, and you should have a very good understanding of how to apply both of these techniques.

I hope that you now understand why I get so excited about springs, and that you have begun to play with them and create some really fun and interesting effects yourself.

Now that you've learned all sorts of ways of moving things around, let's move on to the next chapter, where you'll find out what to do when they start hitting each other!

