

ActionScript 3.0 Cookbook

Programmatic Animation

By Joey Lott, Darron Schall and Keith Peters

Publisher: O'Reilly Media, Inc.

Pub Date: 2006-10-11

ISBN: 9780596526955

Table Of Contents

Programmatic Animation.....	3
Introduction	3
Moving an Object	3
Moving an Object in a Specific Direction	5
Easing	7
Acceleration	9
Springs	10
Using Trigonometry	12
Applying Animation Techniques to Other Properties	15
Index.....	18

Copyright (©) 2004, 2005, 2006, 2007 O'Reilly Media.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472

The copyrights in individual elements of this work are owned by their respective publishers, authors or others, as the case may be, and the prior written permission of the copyright owner is required for reuse in any form or medium of any individual element.

O'Reilly books may be purchased for educational, business or sales promotional use. Online editions are also available for most titles (<http://safari.oreilly.com/>). For more information, contact our corporate/institutional sales department: (800)998.9938 or corporate@oreilly.com.

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. Microsoft, the .NET logo, Virtual C#, Visual Basic, Visual Studio, and Windows are registered trademarks or trademarks of Microsoft Corporation. Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps. The association between the image of the African crowned crane and the topic of C# is a trademark of O'Reilly Media, Inc.

While reasonable care has been exercised in the preparation of this book, the publisher and the authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

Programmatic Animation

Introduction

Animation can be defined as any visual change over time. If an image does not change over a period of time, it's impossible to tell whether it is a still image or an animation. There are a variety of properties you can manipulate to produce change, and thus animation. The most obvious is changing an object's position to make it move. You can also change its size, shape, rotation, transparency or color, to name a few. As long as something changes visually, the viewer never sees the animation.

In the earliest versions of Flash, most animation was done by using *tweens*. An object was placed on a keyframe, another keyframe was made, and the object was changed in some way. Flash filled in the frames *in between*, hence, the term *tween*. Using ActionScript, you can create much more dynamic and interactive animation.

As for what you can animate, a movie clip or sprite is usually a good answer. These objects can contain graphics, and they can have methods and properties that allow them to be moved, scaled, rotated, and otherwise transformed any way you see fit. A movie clip would normally be used only in the Flash authoring environment, where additional frames are added, as in a tween.

Finally, you need some way of getting the changes to occur over time. Your best bet is either an *enterFrame* handler or a timer. ActionScript statements can be used to make changes to the animated object's properties on each frame, or timer cycle, if you're using a timer. Since motion is the most obvious kind of animation, the examples in this chapter start out by moving objects around. As the chapter progresses, you'll see some examples that apply the same techniques to other properties, such as animating the size of an object or its orientation.

Moving an Object

Problem

You have a graphic in a sprite and you want to animate it, giving it some motion.

Solution

Decide on a velocity for the x or y -axis (or both), and add that velocity to the object's position on each frame or animation interval.

Discussion

Velocity is often incorrectly defined as speed. However, velocity also includes a direction factor. For example, "10 miles per hour" is speed, but "10 miles per hour due north" is a velocity. If you

are dealing with velocity on the *x* or *y*-axis, the direction is inherent. A positive *x* velocity is to the right; negative to the left. Likewise, a positive *y* velocity is down, and negative is up.

The first example defines the *x* velocity, *_vx*, and sets it to 3. Since this example uses the *enterFrame* event for animation, the object will move three pixels to the right on each frame:

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

    public class Velocity extends Sprite {
        private var _sprite:Sprite;
        private var _vx:Number = 3;

        public function Velocity( ) {
            _sprite = new Sprite( );
            _sprite.graphics.beginFill(0x0000ff, 100);
            _sprite.graphics.drawCircle(0, 0, 25);
            _sprite.graphics.endFill( );
            _sprite.x = 50;
            _sprite.y = 100;
            addChild(_sprite);
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        public function onEnterFrame(event:Event):void {
            _sprite.x += _vx;
        }
    }
}
```

If you set *_vx* to -3 instead, you'll see that it goes in the opposite direction. You can also add in some *y* velocity by creating a *_vy* variable, giving it a value, and changing the *onEnterFrame* method, as follows:

```
public function onEnterFrame(event:Event):void {
    _sprite.x += _vx;
    _sprite.y += _vy;
}
```

If you aren't a fan of frame-based animation (as in the previous example), you can use a timer function instead, as shown here in bold:

```
package {
    import flash.display.Sprite;
    import flash.events.Event;
    import flash.events.TimerEvent;
    import flash.utils.Timer;

    public class Velocity extends Sprite {
        private var _sprite:Sprite;
        private var _vx:Number = 3;
        private var _vy:Number = 2;
        private var _timer:Timer;

        public function Velocity( ) {
```

```

        _sprite = new Sprite( );
        _sprite.graphics.beginFill(0x0000ff, 100);
        _sprite.graphics.drawCircle(0, 0, 25);
        _sprite.graphics.endFill( );
        _sprite.x = 50;
        _sprite.y = 100;
        addChild(_sprite);
        _timer = new Timer(30);
        _timer.addEventListener("timer", onTimer);
        _timer.start( );
    }

    public function onTimer(event:TimerEvent):void {
        _sprite.x += _vx;
        _sprite.y += _vy;
    }
}

```

See Also

Recipe 11.2 for information on how to move an object at a given speed and angle.

Moving an Object in a Specific Direction

Problem

You want to move an object at a certain speed in a specific angular direction.

Solution

Convert the speed and angle to x and y velocities and add these to the object's x and y position on each frame or animation interval.

Discussion

Recipe 11.1 explains how you can move something at specific velocities on the x and y -axes, but what if you just know the angle and speed you want an object to move? For example, you want the object to move at an angle of 135 degrees, with a speed of 4 pixels per frame.

You can use some basic trigonometric functions to convert this angle and speed to component x and y velocities. First, you need to make sure the angle is in radians. If the angle is in degrees, convert it by using the following formula:

$$\text{radians} = \text{degrees} * \text{Math.PI} / 180;$$

If you ever need to convert the opposite way, use:

$$\text{degrees} = \text{radians} * 180 / \text{Math.PI};$$

Once you have the angle in radians, use the *Math.sin* and *Math.cos* functions, along with the speed, to find the x and y velocities, using the following formulas:

```
vx = Math.cos(angle) * speed;
vy = Math.sin(angle) * speed;
```

Then you can simply move the object as outlined in *Recipe 11.1*. Here is an example using 135 degrees and a speed of 4 pixels per frame:

```
package {
    import flash.display.Sprite;
    import flash.events.Event;
    import flash.events.TimerEvent;
    import flash.utils.Timer;

    public class AngularVelocity extends Sprite {
        private var _sprite:Sprite;
        private var _angle:Number = 135;
        private var _speed:Number = 4;
        private var _timer:Timer;

        public function AngularVelocity () {
            _sprite = new Sprite();
            _sprite.graphics.beginFill(0x0000ff, 100);
            _sprite.graphics.drawCircle(0, 0, 25);
            _sprite.graphics.endFill();
            _sprite.x = 200;
            _sprite.y = 100;
            addChild(_sprite);
            _timer = new Timer(30);
            _timer.addEventListener("timer", onTimer);
            _timer.start();
        }

        public function onTimer(event:TimerEvent):void {
            var radians:Number = _angle * Math.PI / 180;
            var vx:Number = Math.cos(radians) * _speed;
            var vy:Number = Math.sin(radians) * _speed;
            _sprite.x += vx;
            _sprite.y += vy;
        }
    }
}
```

Of course, in such a simple example, it wouldn't make sense to recalculate the x and y velocities on each interval, as they never change. Instead, just calculate it one time, save the result, and use it on each frame. In many cases, however, the speed and direction will be constantly changing, and therefore need to be computed new for each frame.

See Also

Recipe 11.1

Easing

Problem

You want an object to smoothly move to a specific location, slow down, and stop as it reaches that spot.

Solution

Use an easing formula.

Discussion

First, we'll look at the concept of simple *easing*. You have an object at a certain position and you want it to ease to another position. Take the distance between the two points and move the object a fraction of that distance—maybe one-half, one-third, or less. On the next iteration, find the new distance and move the object a fraction of that. Continue this way until the object is so close to the target that you can consider it there.

You'll see that the first couple of jumps are quite big, but successive jumps get smaller and smaller until the object appears not to be moving at all. Viewed in terms of velocity, the velocity starts out high and approaches zero. Another way of looking at it is that velocity is dependent on distance. A large distance makes for a high velocity.

The following example shows a simple example of easing. The target position is specified by `_targetX` and `_targetY`. The fraction that the object moves each time is set in `_easingSpeed`. Here it is set to 0.1, which means the object moves one-tenth of the distance to the target on each animation interval:

```
package {
    import flash.display.Sprite;
    import flash.events.Event;
    import flash.events.TimerEvent;
    import flash.utils.Timer;

    public class Easing extends Sprite {
        private var _sprite:Sprite;
        private var _easingSpeed:Number = 0.1;
        private var _targetX:Number = 400;
        private var _targetY:Number = 200;
        private var _timer:Timer;

        public function Easing() {
            _sprite = new Sprite();
            _sprite.graphics.beginFill(0x0000ff, 100);
            _sprite.graphics.drawCircle(0, 0, 25);
            _sprite.graphics.endFill();
            _sprite.x = 50;
            _sprite.y = 50;
            addChild(_sprite);
            _timer = new Timer(30);
            _timer.addEventListener(TimerEvent.TIMER, onTimer);
            _timer.start();
        }
    }
}
```

```

    }

    public function onTimer(event:TimerEvent):void {
        var vx:Number = (_targetX - _sprite.x) * _easingSpeed;
        var vy:Number = (_targetY - _sprite.y) * _easingSpeed;
        _sprite.x += vx;
        _sprite.y += vy;
    }
}

```

One problem with this setup is that the timer continues to run, even after the object has gotten as close as it's going to get to the target. To handle that, find the distance to the target and if it is less than a certain value, just turn off the timer, as illustrated by the bolded code in the following example:

```

public function onTimer(event:TimerEvent):void {
    var dx:Number = _targetX - _sprite.x;
    var dy:Number = _targetY - _sprite.y;
    var dist:Number = Math.sqrt(dx * dx + dy * dy);
    if(dist < 1)
    {
        _sprite.x = _targetX;
        _sprite.y = _targetY;
        _timer.stop( );
    }
    else
    {
        var vx:Number = dx * _easingSpeed;
        var vy:Number = dy * _easingSpeed;
        _sprite.x += vx;
        _sprite.y += vy;
    }
}

```

This example first finds the distance on the two axes and the total distance. If the distance is less than 1, it places the object at the target point and kills the timer. Otherwise, it continues as normal.

Sometimes though, you may not want the easing to stop; for example, in a moving target. The following example has the object easing toward the mouse. In other words, it simply replaces `mouseX` and `mouseY` for `_targetX` and `_targetY`:

```

public function onTimer(event:TimerEvent):void {
    var vx:Number = (mouseX - _sprite.x) * _easingSpeed;
    var vy:Number = (mouseY - _sprite.y) * _easingSpeed;
    _sprite.x += vx;
    _sprite.y += vy;
}

```

This is the simplest form of easing, and will suffice in a good many cases. Robert Penner (<http://www.robertpenner.com>), a well-known and highly respected Flash programmer, has developed a set of much more complex easing formulas that have become a sort of standard for easing applications. They even have been incorporated into the standard ActionScript classes that come with both Flash and Flex. At this writing, the equations are written for ActionScript 1.0 and 2.0 only, but they could easily be adapted for ActionScript 3.0. These equations do such things as

easing in, easing out, easing in *and* out, or easing based on a specific time interval or number of frames, with many different formulas.

See Also

Recipes *11.1* and *11.2* for the basics on velocity.

Acceleration

Problem

You want an object to start moving slowly and then speed up over time.

Solution

Apply acceleration.

Discussion

Many people think of acceleration as simply speeding up. After all, when you want to go faster in your car, you step on the accelerator. A more scientific definition would be a change in velocity. Although this certainly encompasses increasing an object's speed, it also applies to slowing it down or changing its direction.

Acceleration requires an understanding of velocity as covered in Recipes *11.1* and *11.2*. Acceleration also has a *magnitude* and *direction*, which can be represented as acceleration on the *x* and *y*-axes. With each frame or animation interval, the acceleration of each axis is added to the velocity on that axis, and then the velocity is added to the position, as in *Recipe 11.1*.

The following example uses the variables `_ax` and `_ay` for acceleration and `_vx` and `_vy` for velocity:

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

    public class Accel extends Sprite {
        private var _sprite:Sprite;
        private var _ax:Number = .3;
        private var _ay:Number = .2;
        private var _vx:Number = 0;
        private var _vy:Number = 0;

        public function Accel() {
            _sprite = new Sprite();
            _sprite.graphics.beginFill(0x0000ff, 100);
            _sprite.graphics.drawCircle(0, 0, 25);
            _sprite.graphics.endFill();
            _sprite.x = 50;
            _sprite.y = 100;
            addChild(_sprite);
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }
    }
}
```

```

    public function onEnterFrame(event:Event):void {
        _vx += _ax;
        _vy += _ay;
        _sprite.x += _vx;
        _sprite.y += _vy;
    }
}

```

As you can see, the sprite starts out motionless and gradually picks up speed as it goes across the stage. Generally, since acceleration is additive, the acceleration values should start out small; velocity builds up quickly over time.

Also, similar to *Recipe 11.2*, you can start with a direction and magnitude for the acceleration force:

```

var angle:Number = 45;
var accel:Number = .5;

```

Then convert this to acceleration on each axis:

```

var radians:Number = angle * Math.PI / 180;
_ax = Math.cos(radians) * accel;
_ay = Math.sin(radians) * accel;

```

Now you have values that you can add to the velocity.

It's worth noting that gravity is simply acceleration on the y -axis. You can create a gravity variable and set it to a constant value. Then add it to the y velocity on each frame, and you will have realistic gravity.

See Also

Recipes *11.1* and *11.2* for information on velocity .

Springs

Problem

You want an object to jump to and settle at a specific point, as if it were attached by a spring or rubber band.

Solution

Use *Hooke's Law*, the formula for a spring.

Discussion

Hooke's Law describes the forces at work in a spring. In simple terms, it says that the force applied by the spring (acceleration) is proportional to how far it is stretched. This makes total sense. For example, if you barely pull on a rubber band, it snaps back lightly. But if you pull it back as far as you can, it snaps back with enough force to be painful.

Obviously, springs have different amounts of “springiness” or tension. Some are easy to stretch and won’t snap back too strongly. Others require a lot more force to pull, and will spring back with an equally strong force. A number can be used to represent each spring’s strength. The variable `_k` represents this constant, and it is generally a small fraction of 1. A value such as 0.1 or 0.2 works well.

When a spring is modeled with ActionScript, you also need to specify a target point that the spring will pull the object to. Finally, you need to apply some damping or friction. In the real world, as an object springs back and forth, it loses a bit of energy and eventually comes to rest somewhere. If you don’t add dampening to your code, the object just springs back and forth forever. To apply damping, multiply the velocity values by a fraction, such as 0.95. This removes 5 percent of its speed on each frame, eventually slowing it down to a stop. Here is an example, with all these principles in place:

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

    public class Spring extends Sprite {
        private var _sprite:Sprite;
        private var _vx:Number = 20;
        private var _vy:Number = 0;
        private var _k:Number = .1;
        private var _damp:Number = .94;
        private var _targetX:Number = 200;
        private var _targetY:Number = 200;

        public function Spring() {
            _sprite = new Sprite();
            _sprite.graphics.beginFill(0x0000ff, 100);
            _sprite.graphics.drawCircle(0, 0, 25);
            _sprite.graphics.endFill();
            _sprite.x = 0;
            _sprite.y = 0;
            addChild(_sprite);
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        public function onEnterFrame(event:Event):void {
            var ax:Number = (_targetX - _sprite.x) * _k;
            var ay:Number = (_targetY - _sprite.y) * _k;
            _vx += ax;
            _vy += ay;
            _sprite.x += _vx;
            _sprite.y += _vy;
            _vx *= _damp;
            _vy *= _damp;
        }
    }
}
```

In this example, the spring’s force, represented by the variable `_k`, is set to 0.1. The variable `_damp` is set to 0.94; this variable controls the damping or friction. The target point 200, 200 is stored in the variables `_targetX` and `_targetY`.

In the `onEnterFrame` method, get the distance from the target to the object's current position. This tells you how far the spring is stretched. Multiply this by `_k`, the spring's strength. This gives you the force (or acceleration) to apply. Add this to the velocity and add the velocity to the position.

Finally, apply the damping by multiplying the velocity by the damp variable.

When you test this, you should see the sprite spring quickly to the target point, go past it, and spring back. Eventually it settles down and comes to rest.

The target point does not have to be stationary. You can easily alter the previous example to use the mouse coordinates as a target by changing the two lines that determine the acceleration:

```
var ax:Number = (mouseX - _sprite.x) * _k;
var ay:Number = (mouseY - _sprite.y) * _k;
```

This gives you a very smooth, interactive spring.

Try using different values for `_k` and `_damp` to see how you can alter the spring's properties.

See Also

Recipes [11.1](#), [11.2](#), and [11.4](#) for information velocity and acceleration.

Using Trigonometry

Problem

You want to do some advanced animation, involving rotation, circular motion, or oscillation.

Solution

Use the built-in math functions `Math.sin()`, `Math.cos()`, and `Math.atan2()`.

Discussion

Recipes [11.2](#) and [11.4](#) touched on the use of the sine and cosine functions, but they can be used for many other useful effects, such as moving objects in circular or oval paths, smoothly back and forth around a position, or rotating to a particular angle. Both `Math.sin()` and `Math.cos()` are based on the properties of a right triangle (a triangle that has one 90-degree angle). Without getting into a trigonometry lesson, if you feed either function a series of increasing numbers, they will return values that go smoothly back and forth from -1 to 0, 1, 0, and back to -1, continuously. The following code snippet demonstrates this:

```
for(var i:Number = 0; i < 10; i += 0.1) {
    trace(Math.sin(i));
}
```

This traces a long list of numbers. If you examine those numbers, you'll see that they start at 0, go up to 0.999, back down to -0.999, back up, and so on. You can now multiply that by another number, say 40, and get a list of values from -40 to 40. If you use this in an `enterFrame` handler, or timer-based method, and apply the result to an object's position, you can get it to oscillate back and forth, or up and down, as the following example shows:

```

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

    public class Oscillation extends Sprite {
        private var _sprite:Sprite;
        private var _angle:Number = 0;
        private var _radius:Number = 100;

        public function AS3CB( ) {
            _sprite = new Sprite( );
            _sprite.graphics.beginFill(0x0000ff, 100);
            _sprite.graphics.drawCircle(0, 0, 25);
            _sprite.graphics.endFill( );
            _sprite.x = 0;
            _sprite.y = 100;
            addChild(_sprite);
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        public function onEnterFrame(event:Event):void {
            _sprite.x = 200 + Math.sin(_angle) * _radius;
            _angle += .05;
        }
    }
}

```

Here, `_angle` is the variable holding the increasing value fed to `Math.sin()`. The result is multiplied by the `_radius` variable, which is set at 100. This causes the sprite to go back and forth 100 pixels.

If you use `Math.cos()` and do the same thing with the sprite's `y` position, you have circular motion:

```

public function onEnterFrame(event:Event):void {
    _sprite.x = 200 + Math.sin(_angle) * _radius;
    _sprite.y = 200 + Math.cos(_angle) * _radius;
    _angle += .05;
}

```

To make more of an oval shaped path, just use a different radius value on each axis. For example, set `_xRadius` to 100, `_yRadius` to 50, and do the following:

```

public function onEnterFrame(event:Event):void {
    _sprite.x = 200 + Math.sin(_angle) * _xRadius;
    _sprite.y = 200 + Math.cos(_angle) * _yRadius;
    _angle += .05;
}

```

Now, if you create separate angles and amounts to add to each angle, you can get a very random-looking motion. First, create separate variables for each axes' factors:

```

private var _xAngle:Number = 0;
private var _yAngle:Number = 0;
private var _xSpeed:Number = .13;
private var _ySpeed:Number = .09;
private var _xRadius:Number = 100;
private var _yRadius:Number = 50;

```

Then apply those to the motion code:

```
public function onEnterFrame(event:Event):void {
    _sprite.x = 200 + Math.sin(_xAngle) * _xRadius;
    _sprite.y = 200 + Math.cos(_yAngle) * _yRadius;
    _xAngle += _xSpeed;
    _yAngle += _ySpeed;
}
```

One possible use for this example is to simulate a fly, randomly buzzing around a room.

Another very useful trig function is *Math.atan2()*. The main use for this is in finding the angle between two points. It takes two parameters: the distance between the two points on the *y*-axis, and the distance between them on the *x*-axis. It then returns the angle, in radians, between the points.

A common scenario for using *Math.atan2()* is in making an object (a sprite, for example) point at the mouse. The *y* distance is `mouseY - _sprite.y`, and the *x* distance is `mouseX - _sprite.x`. *Math.atan2()* returns an angle. Convert that to degrees and use it to set `_sprite.rotation`. Of course, you'll need some sort of sprite graphic that shows which direction it is rotating. The next example creates the classic "following eyes," a little desktop toy that follows the mouse around the screen and has been created for just about every graphical operating system out there (actually, this example creates only a single eye, but it demonstrates the principle):

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

    public class FollowingEye extends Sprite {
        private var _sprite:Sprite;

        public function AS3CB() {
            _sprite = new Sprite();
            _sprite.graphics.beginFill(0xffffffff, 100);
            _sprite.graphics.drawCircle(0, 0, 25);
            _sprite.graphics.endFill();
            _sprite.graphics.beginFill(0x000000, 100);
            _sprite.graphics.drawCircle(20, 0, 5);
            _sprite.graphics.endFill();
            _sprite.x = 100;
            _sprite.y = 100;
            addChild(_sprite);
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        public function onEnterFrame(event:Event):void {
            var dx:Number = mouseX - _sprite.x;
            var dy:Number = mouseY - _sprite.y;
            var radians:Number = Math.atan2(dy, dx);
            _sprite.rotation = radians * 180 / Math.PI;
        }
    }
}
```

The setup code draws an extra circle on the right edge of the first. When you are doing this kind of rotation, align your graphics so that “zero degrees” is facing to the right like this. The *enterFrame* handler calculates the two distances and the resulting angle, converts to degrees and assigns it to the eye’s rotation.

Applying Animation Techniques to Other Properties

Problem

You want to apply the techniques in this chapter’s recipes to something other than an object’s motion.

Solution

Apply the techniques as given, but assign the results to a property other than the object’s *x* and *y* position.

Discussion

Although changing an object’s position is the most obvious method of animation, all of the techniques in this chapter can be applied to almost any property of a movie clip or sprite. This recipe provides several examples to get you started, but the possibilities are so numerous that it would be impossible to list them all.

First, try applying some velocity to the rotation property; this variable is called *_vr* for rotational velocity:

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

    public class AnimatingRotation extends Sprite {
        private var _sprite:Sprite;
        private var _vr:Number = 4;

        public function AS3CB() {
            _sprite = new Sprite();
            _sprite.graphics.beginFill(0xffffff, 100);
            _sprite.graphics.drawRect(-50, -20, 100, 40);
            _sprite.graphics.endFill();
            _sprite.x = 100;
            _sprite.y = 100;
            addChild(_sprite);
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
        }

        public function onEnterFrame(event:Event):void {
            _sprite.rotation += _vr;
        }
    }
}
```

This example uses a rectangle instead of a circle, so you can see the rotation in action. It sets `_vr` to 4, and then adds that to the sprite's rotation on each frame.

The next example applies a spring formula to the scale of the sprite. A click handler sets a random target scale, and the `enterFrame` handler applies the spring action. When you click on the sprite, it bounces to a new size:

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

    public class AnimatingProperties extends Sprite {
        private var _sprite:Sprite;
        private var _k:Number = 0.1;
        private var _damp:Number = 0.9;
        private var _scaleVel:Number = 0;
        private var _targetScale:Number = 1;

        public function AS3CB() {
            _sprite = new Sprite();
            _sprite.graphics.beginFill(0xffffffff, 100);
            _sprite.graphics.drawRect(-50, -50, 100, 100);
            _sprite.graphics.endFill();
            _sprite.x = 100;
            _sprite.y = 100;
            addChild(_sprite);
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
            _sprite.addEventListener(MouseEvent.CLICK, onClick)
        }

        public function onEnterFrame(event:Event):void {
            _scaleVel += (_targetScale - _sprite.scaleX) * _k
            _sprite.scaleX += _scaleVel;
            _sprite.scaleY = _sprite.scaleX;
            _scaleVel *= _damp;
        }

        public function onClick(event:MouseEvent):void {
            _targetScale = Math.random() * 2 - .5;
        }
    }
}
```

You could create similar functionality using easing (see *Recipe 11.3*) instead of springs. Photo gallery applications often use this technique for displaying differently sized photos. The photo content area eases to the new size, and then the photo fades in.

A more complex application of motion code is to use the techniques to color transforms to smoothly go from one color to another. This is probably best done with easing. Start out with one color and gradually ease into another color.

The following example sets up two sets of color values: `_red1`, `_green1`, `_blue1`, and `_red2`, `_green2`, `_blue2`. Each value is a number from 0.0 to 1.0. In the `enterFrame` handler, these values are fed to the red, green, and blue multiplier values of a *color transform* object that is applied to

the sprite. All of the values ease from the first to the second value, so they smoothly transition from one to the other. There is also a click handler on the sprite, which randomly sets three new multiplier values; each time you click on the square, it eases to a new color:

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

    public class AnimatingColor extends Sprite {
        private var _sprite:Sprite;

        private var _red1:Number = 1;
        private var _green1:Number = 0;
        private var _blue1:Number = 0;

        private var _red2:Number = 0;
        private var _green2:Number = .5;
        private var _blue2:Number = 1;

        private var _easingSpeed:Number = 0.05;

        public function AS3CB() {
            _sprite = new Sprite();
            _sprite.graphics.beginFill(0xffffffff, 100);
            _sprite.graphics.drawRect(-50, -50, 100, 100);
            _sprite.graphics.endFill();
            _sprite.x = 100;
            _sprite.y = 100;
            addChild(_sprite);
            addEventListener(Event.ENTER_FRAME, onEnterFrame);
            addEventListener(MouseEvent.CLICK, onClick);
        }

        public function onEnterFrame(event:Event):void {
            _red1 += (_red2 - _red1) * _easingSpeed;
            _green1 += (_green2 - _green1) * _easingSpeed;
            _blue1 += (_blue2 - _blue1) * _easingSpeed;
            _sprite.transform.colorTransform =
                new ColorTransform(_red1, _green1, _blue1);
        }

        public function onClick(event:MouseEvent):void {
            _red2 = Math.random();
            _green2 = Math.random();
            _blue2 = Math.random();
        }
    }
}
```

See Also

Recipe 11.3

Index

A

acceleration of display objects, 9
 springs, 10
animation (programmatic), 3
 applying techniques to other properties, 15
 moving objects, 3
 specific directions, 5, 6

C

color transform objects, 16

D

direction (acceleration), 9

E

easing, moving display objects, 7
enterFrame event, 3, 16

H

Hooke's Law, 10

K

keyframes, 3

M

magnitude (acceleration), 9
Math.atan2() function, 12
Math.cos() function, 12
Math.sin() function, 12

O

objects
 acceleration of, 9
 easing, 7
 moving, 3
 springs, 10

P

programmatic animation, 3

S

speed of display objects, 3
springs, 10

T

trigonometry, 12
tweens, 3

V

velocities for display objects, 3

X

x-axis, moving objects, 3

Y

y-axis, moving objects, 3