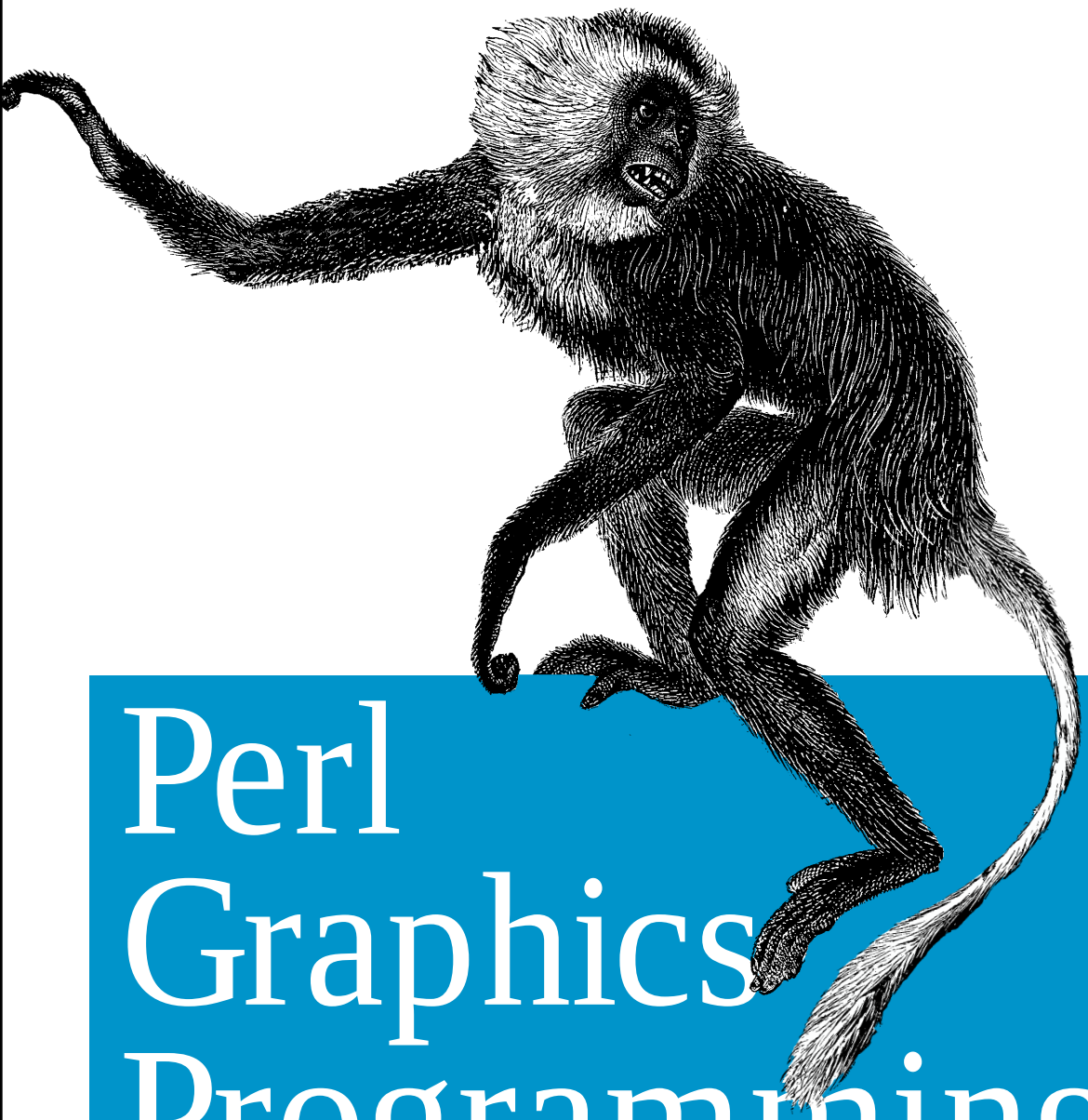
Creating SVG, SWF (Flash), JPEG, and PNG files with Perl



Perl Graphics Programming

O'REILLY®

Shawn Wallace

Perl Graphics Programming

Shawn Wallace

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

CHAPTER 9

Using Ming

Ming is a library written in C that provides a simple API for creating SWF files (described in Chapter 8). In addition to the Perl wrapper described in this chapter, Ming comes with interfaces to PHP, Python, Ruby, Java, and C++. The Ming library was created by Dave Hayden of Opaque Industries and is released under the LGPL (Lesser GNU Public License).

Most web-graphics designers create SWF files with Macromedia's Flash software, which provides a GUI and tools for drawing and manually piecing together the timeline of a movie. With Ming, you can create an assembly line of scripts that automatically update the SWF content of your web site. You can also use Ming with CGI scripts to dynamically generate SWF documents from various data sources. In some cases, it may even make sense to use Ming instead of the Flash tool, or to supplement Flash with Ming at various places in the workflow.

This chapter is divided into three sections. The first introduces a comprehensive Flash application (a game called Astral Trespassers) that touches on all of the basic Ming objects. Next is a reference guide to Ming's Perl interface. The last section of the chapter provides solutions to some problems you may encounter when using Ming. By the end of the chapter you will be able to draw circles using cubic Bezier curves, attach a preloader to your movie, and communicate between an SWF client and a Perl backend using sockets.

Installation

To install Ming, download the source distribution from *http://www.opaque.net/ming*. Ming was designed to not rely on any external libraries, which complicates some aspects of using the library. For example, fonts and images must be converted into special Ming-readable exchange formats instead of being read directly (using a library such as *libpng* or Freetype). The lack of dependencies makes for a simple installation, though; everything is set up from a preconfigured Makefile. The library should work on most platforms, including Windows.

The Perl wrapper, an XS interface to the C library written by Soheil Seyfaie, can be installed after the library is installed. The Perl interface comes with the standard Ming distribution, and has been successfully tested on Unix and MacOS-based systems.

Overview of the Perl Interface

The Perl interface consists of 14 modules in the SWF namespace:

SWF::Movie
: This module implements a top-level timeline as a Movie object.

SWF::Sprite
: Also called a MovieClip, a Sprite object is an encapsulated movie that can be combined with other Sprites on a Movie timeline.

SWF::DisplayItem
: Once an object has been placed on the Stage of a Movie, the instance of that object can be manipulated as a DisplayItem.

SWF::Shape
: This is a reusable vector shape object.

SWF::Button
: A button is an object that defines shapes within a frame that can respond to mouse and keyboard events.

SWF::Bitmap
: An image can be read from a file and placed as a Bitmap object within the Movie.

SWF::Text
: The Text object allows you to draw strings on a frame.

SWF::TextField
: A TextField can be used for interactive forms or for boxes of text that need to be dynamically updated.

SWF::Font
: A Font object describes the glyph shapes used to draw Text and TextField objects.

SWF::Fill
: This is a description of one of a variety of fills that can be associated with a Shape.

SWF::Gradient
: This is a description of a gradient that can be used to create a Fill object.

SWF::Morph
: A Morph object represents the gradual transition between two shapes.

SWF::Sound
: An MP3 can be read from a file and placed on the timeline as a Sound object.

SWF::Action
: This is an object containing a block of ActionScript that can be associated with a timeline or another object.

The following example features most of Ming's object types. The example is followed by a function reference for each of the Perl modules.

The Astral Trespassers Game

Astral Trespassers is a simple shoot-em-up space game. When executed, it generates a self-contained SWF file that implements the game shown in Figure 9-1. The player uses the mouse to control a gun positioned at the bottom of the screen—if the mouse button is clicked within the frame, a photon bullet is fired. The player is responsible for destroying a phalanx of aliens (represented by primitive blue SWF::Shape objects) that advance down the screen. This example will illustrate the use of the Ming API to create simple shapes, attach ActionScript to buttons and frames, and manipulate multiple Sprites. It also provides a rough template for your own interactive applications.

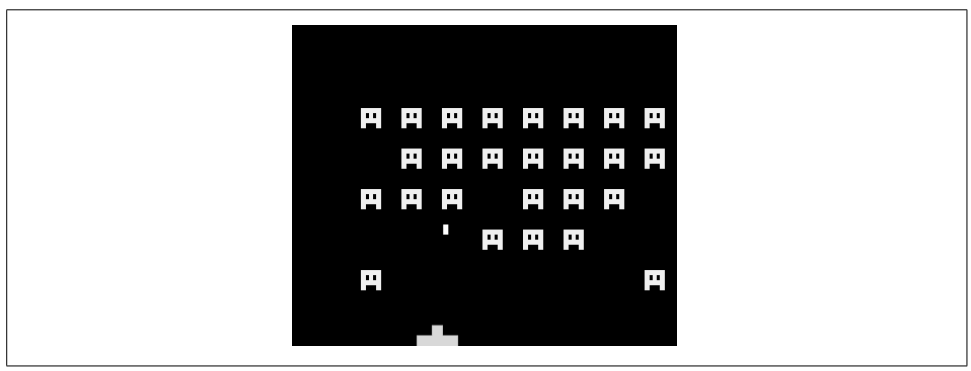


Figure 9-1. Astral Trespassers, a sample application created using the Ming SWF library

The document is created in five stages:

1. Create the SWF::Shape objects that are used to build the Sprites for the gun, aliens, and photon bullet.
2. Assemble the Shapes into Sprites.
3. Assemble the Sprites on the main timeline of the SWF::Movie object.
4. Attach ActionScript to individual Sprites and the main timeline. The ActionScript is interpreted by the Flash player and controls the movement of sprites on the Stage and user interaction when the movie is played back.
5. Write the Movie to a file (or STDOUT) formatted as SWF data.

The SWF document (we'll call it *astral_trespassers.swf*) is about 2k in size. The following code loads the modules and sets the default scale:

```
#!/usr/bin/perl -w
#
# Astral Trespassers

use strict;
use SWF qw(:ALL);             # Import all of the Ming modules

SWF::useSWFVersion(5);        # This script uses some Flash 5 ActionScript

# Set the scale. 20 is the default, which means that
# all scalar values representing coordinates or dimensions
# are indicated in twips (1/20th of a pixel).

SWF::setScale(20);
```

Now we must create the shapes that make up the Sprites. The player's gun consists of two red rectangles, where the origin of the gun is centered on the barrel. The `drawRect( )` function is defined at the end of this script and adds a rectangular path to the shape. The arguments are a Shape object, *`width`*, *`height`*, *`dx`*, and *`dy`*. The upper left corner of the rectangle is drawn shifted by *`dx`*,*`dy`* from the origin of the shape.

```
my $s = new SWF::Shape( );
$s->setLeftFill($s->addFill(255, 0, 0));
drawRect($s, 10, 10, -5, 0);
drawRect($s, 40, 10, -20, 10);
```

Now we'll create an alien, with the origin of each centered between its legs. Each alien will be a Sprite object consisting of two frames to simulate the movement of its legs. Let's define the two shapes here.

```
my $s2 = new SWF::Shape( );
$s2->setLeftFill($s2->addFill(0, 0, 255));
drawRect($s2, 20, 15, -10, -15);    # The body
drawRect($s2, 5, 5, -10, 0);        # Left leg
drawRect($s2, 5, 5, 5, 0);          # Right leg
drawRect($s2, 3, 5, -5, -10);       # Left eye
drawRect($s2, 3, 5, 2, -10);        # Right eye

my $s3 = new SWF::Shape( );
$s3->setLeftFill($s3->addFill(0, 0, 255));   # Blue
drawRect($s3, 20, 15, -10, -15);    # Body
drawRect($s3, 5, 5, -7, 0);         # Left leg
drawRect($s3, 5, 5, 2, 0);          # Right leg
drawRect($s3, 3, 5, -5, -10);       # Left eye
drawRect($s3, 3, 5, 2, -10);        # Right eye
```

The photon bullet is a simple white rectangle:

```
my $s4 = new SWF::Shape( );
$s4->setLeftFill($s4->addFill(255, 255, 255));
drawRect($s4, 5, 10, -3, -10);
```

Finally, we need to define a shape for the button that collects the player's mouse clicks. The button has the same dimensions as the movie:

```
my $s5 = new SWF::Shape( );
$s5->setRightFill($s5->addFill(0, 0, 0));
drawRect($s5, 400, 400, 0, 0);
```

Movies can be nested within each other. The SWF::Sprite module represents a movie clip that has its own timeline and can be manipulated as a discrete object on the root movie timeline. The next step is to create all of the sprites needed in the movie. Start with the player's gun, which is one frame long:

```
my $gun = new SWF::Sprite( );
$gun->add($s);
$gun->nextFrame( );
```

Next, create the alien sprite, which is four frames long: two with shape `$s2` and two with shape `$s3`. The `$item` object is of type SWF::DisplayItem, returned when the sprite is added to the movie clip's timeline.

```
my $alien = new SWF::Sprite( );
my $item = $alien->add($s2);
$alien->nextFrame( );
$alien->nextFrame( );
$alien->remove($item);
$alien->add($s3);
$alien->nextFrame( );
$alien->nextFrame( );
```

The photon bullet is another single-frame sprite:

```
my $bullet = new SWF::Sprite( );
$bullet->add($s4);
$bullet->nextFrame( );
```

Next, create a TextField object for the "Game Over" message. Note that the font definition file (see the "The SWF::Font Module" section later in this chapter) called *serif.fdb* must be in the same directory as this script.

```
my $f = new SWF::Font("serif.fdb");
my $tf = new SWF::TextField( );
$tf->setFont($f);
$tf->setColor(255,255,255);
$tf->setName("message");
$tf->setHeight(50);
$tf->setBounds(300,50);
```

Next we will assemble these pieces on the main timeline. The final movie timeline consists of four frames:

Frame 1
: Contains ActionScript that initializes variables used to move aliens and bullets.

Frame 2
: Contains the "hit region" button, the gun sprite, 40 alien sprites, and an off-screen bullet sprite.

Frame 3
: Contains ActionScript that moves the gun, the aliens, and the bullet, and checks to see if a bullet has collided with an alien. If an alien reaches the bottom of the screen, the movie stops.

Frame 4
: Creates an event loop by returning to the previous frame.

Some objects have ActionScript code attached to them. ActionScript is a JavaScript-like interpreted language that is parsed and executed by the SWF player. In our example, ActionScript is attached to the first, third, and fourth frames of the movie, and also to a screen-sized button that is used to gather the user's mouse clicks. Figure 9-2 shows the various objects on the main timeline and the ActionScript (if any) that is attached to each object.

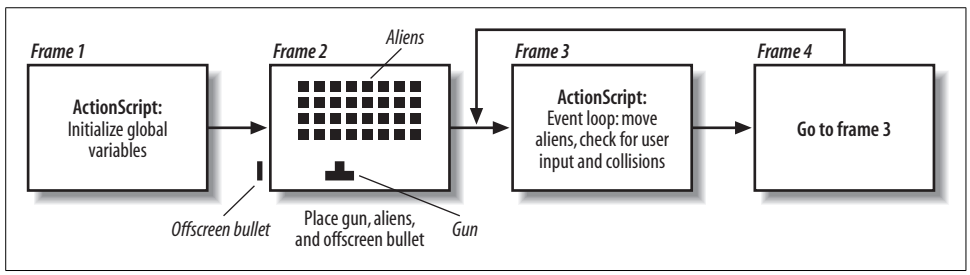


Figure 9-2. The main timeline of the Astral Trespassers example

Now, we'll create a new Movie object:

```
my $m = new SWF::Movie;
$m->setDimension(400, 400);
$m->setBackground(0, 0, 0);
$m->setRate(16);              # Frames per second.
```

The first frame is a keyframe that initializes two variables. The `direction` array keeps track of whether each alien is moving left (–1), right (1), or not moving (0, after it has been shot). `onScreenBullet` is a Boolean indicating whether the bullet is on the screen.

```
$m->add(new SWF::Action(<<ENDSCRIPT
    direction = new Array( );
    for (i=0; i<40; i++) {
        direction[i] = 1;
    }
    onScreenBullet = 0;
ENDSCRIPT
));
$m->nextFrame( );
```

In the second frame we add the sprites to the Stage. First we add the button that acts as a "hot spot" for collecting mouse clicks. The user must click in the area defined by

the shape $s5. Note that the first click gets the button's focus, and subsequent clicks fire bullets.

```
my $b = new SWF::Button( );
$b->addShape($s5, SWF::Button::SWFBUTTON_HIT);

# Add Actionscript that is activated when the button is pressed.

$b->addAction(new SWF::Action(<<ENDSCRIPT
if (onScreenBullet == 0) {
    onScreenBullet = 1;
    _root["bullet"]._x = _root["gun"]._x;
    _root["bullet"]._y = _root["gun"]._y;
}
ENDSCRIPT
), SWF::Button::SWFBUTTON_MOUSEDOWN);

$item = $m->add($b);         # Add the button to the Stage
$item = $m->add($tf);        # Add the "Game Over" text field
                             # Initially it is empty
$item->moveTo(75, 100);
$item = $m->add($gun);       # Add the gun to the Stage
$item->setName("gun");       # Label the gun for use in later
                             # Actionscripts
$item->moveTo(200, 380);     # Position the gun at the bottom
                             # of the screen
foreach my $row (1..5) {     # Add a phalanx of 40 aliens
    foreach my $col (1..8) {
        $item = $m->add($alien);              # Add alien
        $item->moveTo(40*$col, 40*$row);      # Position alien
        $item->setName("alien".               # Label alien
                       (($row-1)*8+$col-1));  # (e.g. 'alien1')
    }
}

$item = $m->add($bullet);      # Add bullet to stage
$item->moveTo(-10, -10);       # Position it off screen
$item->setName("bullet");      # Label it
$m->nextFrame( );
```

Add ActionScript to the third frame for the event loop. If you've never used ActionScript before, Appendix D is an ActionScript reference geared toward Ming developers.

The movement of the gun, aliens, and bullet are all controlled by the following bit of ActionScript that is executed every time Frame 3 is executed. The script moves the player's gun, moves each of the aliens, checks for a collision with the bullet, the edge of the screen, or the bottom of the screen, and moves the bullet:

```
$m->add(new SWF::Action(<<ENDSCRIPT
    /* Move the gun to follow the mouse. Note that the
       gun moves faster the farther it is from the mouse */
    dx = int(_xmouse - _root["gun"]._x)/10;
```

```
    xPos = _root["gun"]._x + dx;
    if ((xPos > 0) && (xPos < 400)) {
        _root["gun"]._x += dx;
    }

    /* Move each of the aliens */
    for (i=0; i<40; i++) {
        /* If an alien reaches the bottom, end the game */
    if (_root["alien"+i]._y > 380) {
            message = "Game Over";
            stop();
    }

    /* If an alien hits one of the margins, reverse direction */
    if (_root["alien"+i]._x > 380) {
            direction[i] = -1;
            _root["alien"+i]._y += 20;
            }
        if (_root["alien"+i]._x < 20) {
            direction[i] = 1;
            _root["alien"+i]._y += 20;
        }

        /* Move the alien */
        _root["alien"+i]._x += direction[i] * 5;

        /* Check to see if the bullet has collided with the alien
           If so, move the bullet and alien off screen */
        if (onScreenBullet & _root["bullet"].hitTest(_root["alien"+i])) {
            _root["bullet"]._y = -10;
            _root["alien"+i]._y = -10;
            onScreenBullet = 0;
            direction[i] = 0;
        }
    }

    /* If the bullet is on the screen, move it upward. */
    if (onScreenBullet) {
        _root["bullet"]._y -= 10;
        if (_root["bullet"]._y < 0) {
            onScreenBullet = 0;
        }
    }
ENDSCRIPT
));
```

The fourth frame closes the event loop by returning to Frame 3:

```
$m->nextFrame();
$m->add(new SWF::Action(<<ENDSCRIPT
    prevFrame();
    play();
ENDSCRIPT
));
$m->nextFrame();
```

Write the result to a file with the save( ) method:

```
$m->save("astral_trespassers.swf");\
exit( );
```

drawRect( ) is a helper procedure used to draw rectangles within a shape:

```
sub drawRect {
    my $shape = shift;
    my ($w, $h, $dx, $dy) = @_;
    $shape->movePenTo($dx, $dy);
    $shape->drawLine($w, 0);
    $shape->drawLine(0, $h);
    $shape->drawLine(-$w, 0);
    $shape->drawLine(0, -$h);
}
```

This script could easily be converted to a CGI script by outputting the appropriate HTTP header and using the output( ) method instead of the save( ) method.

The SWF Module

The top-level SWF module implements only two functions; the rest are defined by the other modules described below. By default, the SWF module does not import any of the other modules. You can import all of them with the directive:

```
use SWF qw(:ALL);
```

or you can import just the modules you intend to use by supplying a list of module names:

```
use SWF qw(Movie Text Font);
```

The following two functions affect global properties of the generated SWF movie.

setScale( )

SWF::setScale(*scale*)

By default, all coordinates and dimensions supplied to any of the functions described in this chapter are expressed in twips, or 1/20 of a pixel (see Chapter 8 for more on the SWF coordinate system). The setScale( ) function lets you write scripts using different scales. For example:

```
SWF::setScale(1);
```

indicates that all of the scalar coordinates and dimensions specified in the script represent actual pixels in the resulting movie.

Note that the scale defined by this function does not apply to coordinates in ActionScript strings. ActionScript code should always represent coordinates using twips.

setVersion( )

```
SWF::setVersion(version)
```

This command indicates the latest version of the SWF specification to which the file conforms. In other words, if the file contains ActionScript syntax that was defined in Version 5.0 of the SWF specification, you should indicate that with the command:

```
SWF::setVersion(5);
```

If you specify Version 4 (4 and 5 are the only options), all of your ActionScript is parsed by a Version 4.0 interpreter, and you get an error if non-4.0 code is encountered.

The SWF::Movie Module

The Movie object defines the top-level timeline of an SWF document. To create a movie clip that can be included as an object on a timeline with other movie clips, use the SWF::Sprite module. Many of the movie methods apply to sprites also.

add( )

```
$displayItem = $movie->add(object);
```

The `add( )` method places an object on the movie's Stage (the current frame). The object can be an SWF::Sprite, SWF::Shape, SWF::Button, SWF::Bitmap, SWF::Text, SWF::TextField, SWF::Morph, or SWF::Sound. A new DisplayItem object is returned, which may be used to position or manipulate the object within the frame, as described later in the "The SWF::DisplayItem Module" section. The same object can be added multiple times to the same movie, where a new DisplayItem is created for each addition.

addAction( )

```
$movie->addAction(action)
```

This method adds ActionScript code to the current frame of the movie. The `action` parameter is an SWF::Action object. The script is executed when the frame is encountered in the timeline.

labelFrame( )

```
$movie->labelFrame(label)
```

This method associates a name with the current frame. The label can be used to identify this frame in an SWF::Action script (see the later section "The SWF::Action Module").

new( )

```
$movie = new SWF::Movie( );
```

This method creates a new Movie object.

nextFrame( )

```
$movie->nextFrame( )
```

When you are finished constructing a frame of a movie, use the `nextFrame( )` method to move on to the next frame. A new frame is added to the movie. Note that all of the objects present in the previous frame remain on the Stage in the next frame, and DisplayItems may still be used to access items within the frame.

output( )

```
$movie->output( )
```

This method prints the movie to STDOUT as a formatted SWF file. Remember to use `binmode( )` on systems where that is necessary.

If you are sending the output directly to a web browser as part of a CGI script, print the HTTP header for an SWF file first:

```
print "Content-type: application/x-shockwave-flash\n\n";
$movie->output( );
```

remove( )

```
$movie->remove(displayItem)
```

This method removes a particular item from the display list. The item no longer appears in the current frame.

save( )

```
$movie->save(filename)
```

This method attempts to open the file with the given name (using `fopen( )`) and writes the SWF document to the file.

setBackground( )

```
$movie->setBackground(red, green, blue);
```

This function sets the background color of the movie. Specify the RGB components of the color as numbers between 0 and 255.

setDimension( )

```
$movie->setDimension(width, height)
```

This method sets the width and height (expressed in twips) of the movie. The rendered dimensions of the movie are subject to the scaling proportion set with `SWF::SetScale( )`.

setFrames( )

```
$movie->setFrames(frames)
```

This method sets the total number of frames in the movie.

setRate( )

```
$movie->setRate(frameRate);
```

The playback rate for the movie can be controlled with this method. The given frame rate (expressed in frames per second) is the maximum frame rate; the SWF player may slow the movie down or skip frames if it can't render the frames fast enough.

setSoundStream( )

```
$movie->setSoundStream(sound)
```

This method adds an MP3 sound to the SWF file at the current frame. The *sound* parameter should be an SWF::Sound object, which represents an MP3 file to be embedded in the SWF document. See the later section "The SWF::Sound Module" for more on sounds.

The SWF::Sprite (or SWF::MovieClip) Module

The SWF specification defines an object called a Sprite, an encapsulated movie with its own timeline that can be added to another movie's timeline. The sprite plays concurrently with any other objects or MovieClips on the main timeline. Flash users may be more familiar with the term MovieClip; you can use the alias SWF::MovieClip for a sprite if you want to.

The behavior of SWF::Sprite methods is identical to that of the methods described for the SWF::Movie object. The following methods may be called on a sprite:

```
new( )
add( )
remove( )
nextFrame( )
setFrames( )
labelFrame( )
```

Remember that the `add()` method returns a DisplayItem that refers to the object on the sprite's timeline, not on the parent movie's timeline. See the Astral Trespassers script at the beginning of this chapter for an example of an application using multiple sprites.

Several ActionScript commands operate on sprites, but ActionScript uses the term MovieClip instead of sprite. For example, the `duplicateMovieClip()` ActionScript method can be applied to a sprite to create a new copy of itself. See Appendix D for a complete ActionScript command reference.

The SWF::DisplayItem Module

Each object that is on-screen at any particular time has an entry in a data structure in the SWF player called the *display list*. The display list keeps track of the position of the object, the depth of the object, and a transformation matrix that affects how the object is drawn on the screen. The SWF::DisplayItem object defines methods for moving, transforming, and arranging objects in the display list. The following attributes are contained in a DisplayItem:

Name
: A label used to refer to the item in ActionScript scripts

Position
: The x, y coordinate of the item within a frame

Scale
: A horizontal and vertical scale multiplier

Rotation
: An angular offset

Skew
: A horizontal and vertical skew offset

Depth
: The position of the item in the display list

Ratio
: If the displayed item is a Morph object, the ratio attribute determines which frame of the morph transition is displayed

Color transform
: The object's red, green, blue, and alpha components may have a color offset applied to them

DisplayItems do not have their own constructors. New DisplayItems are created when a displayable object is added to a movie or a sprite; the `add()` method of these objects returns an SWF::DisplayItem object.

In the example at the beginning of this chapter, we moved the various items around the Stage using an ActionScript event loop. Example 9-1 creates a 60-frame animation where each shape is manually placed within the frame by maintaining a list of DisplayItems for each object on the Stage.

Example 9-1. Using SWF::DisplayItem to position each element of each frame

```
#!/usr/bin/perl -w
#
# Example 9-1. A grid of red squares collapses in on itself,
# then expands to its original state.

use strict;
use SWF qw(Movie Shape DisplayItem);

SWF::setScale(1.0);

# Define a grid

my $grid = 8;
my ($w, $h) = ($grid*100, $grid*100);

# Create a square

my $s = new SWF::Shape( );
$s->setLineStyle(1, 255, 0, 0);
$s->movePenTo(0,0);
$s->drawLineTo(0, 50);
$s->drawLineTo(50, 50);
$s->drawLineTo(50, 0);
$s->drawLineTo(0, 0);

# The displayList array holds the DisplayList objects as they are placed onstage

my @displayList = ( );
my $m = new SWF::Movie( );
$m->setDimension($w, $h);

# Place a grid of squares on the stage and store the reference to each DisplayItem

foreach my $i (0..$grid-1) {
    foreach my $j (0..$grid-1) {
        my $item = $m->add($s);
        $item->moveTo($i*100, $j*100);
        push @displayList, $item;
    }
}

# Now create 30 frames; in each frame, move each square
# 1/30th of the way toward the center of the grid, and rotate
# the square 360/30 degrees. Then repeat the same thing in the
# opposite direction, ending up where we started.
```

Example 9-1. Using SWF::DisplayItem to position each element of each frame (continued)

```
my $frames = 30;
my ($cx, $cy) = ($w/2, $h/2);
foreach my $direction (1, -1) {          # 1 =in, -1 = out
    foreach my $f (1..$frames) {
        foreach my $i (0..$grid-1) {
            foreach my $j (0..$grid-1) {
                $displayList[$i*$grid+$j]->move(
                    $direction*(int($cx-$i*100)/$frames),
                    $direction*(int($cy-$j*100)/$frames));
                $displayList[$i*$grid+$j]->rotate(360/$frames);
            }
        }
        $m->nextFrame( );
    }
}

# Create the SWF file

$m->save("example9-1.swf");
```

Note that the resulting file is quite large for a Flash movie (around 68k). In this particular example, you would be better off moving each square using ActionScript rather than creating static frames for each step of the animation.

move( )

`$displayItem->move(`*dx*`, `*dy*`)`

This function moves the origin of the specified item to a new coordinate that is offset from its current position by (*dx*, *dy*).

moveTo( )

`$displayItem->moveTo(`*x*`, `*y*`)`

This function moves the origin of the specified item to the given (*x*, *y*) global coordinate. The item remains at the same depth within the display list.

remove( )

`$displayItem->remove( )`

This method removes the specified item from the display list with which it is associated. Same as `SWF::Movie::remove( )`.

rotate( )

```
$displayItem->rotate(degrees)
```

This method adds the specified number of degrees to the DisplayItem's current rotation value.

rotateTo( )

```
$displayItem->rotateTo(degrees)
```

This method sets the rotation attribute of the DisplayItem (initially 0), expressed in degrees. When drawn on the frame, the object is rotated around its origin by the given amount.

scale( )

```
$displayItem->scale(x_scale, y_scale)
```

This method scales the object like `scaleTo( )`, but multiplies the current scale by the given values.

scaleTo( )

```
$displayItem->scaleTo(x_scale, y_scale)
```

Each DisplayItem has a scaling attribute that is initially set to 1, indicating that the object should be drawn on the frame using the dimensions with which it was originally defined. The `scaleTo( )` function sets the horizontal and vertical scale to the specified values, replacing the current scale value. Scaling an object affects the object's local coordinate space, so line widths are scaled along with any objects positioned inside the scaled object (if it is a sprite). If `scaleTo( )` is called with only one value, the object is scaled proportionately.

setColorAdd( )

```
$displayItem->addColor(red, green, blue [,alpha])
```

This method adds the given values to the color components of the item. If the item is a sprite, all of the objects within the sprite have the color transform applied to them also.

setColorMult( )

```
$displayItem->multColor(red, green, blue [,alpha])
```

This method multiplies each of the color components of the item by the given values. Component values greater than 255 are taken to be 255.

setDepth( )

```
$displayItem->setDepth(depth)
```

This method sets the depth of the item in the display list. Each item displayed in a frame has its own unique depth value that determines the order in which it is drawn on the screen.

setName( )

```
$displayItem->setName(name)
```

This method labels the item in the display list with the given name. This name can be used to identify the item in ActionScript code.

setRatio( )

```
$displayItem->setRatio(ratio)
```

The *ratio* attribute applies only to SWF::Morph items. The ratio is a real number between 0 and 1.0 that represents a point in the transformation between the two extremes of the morph. Setting the ratio to .5, for example, causes the shape that is halfway between the morph's extremes to be displayed. See the example code under "The SWF::Morph Module" for an example of this.

skewX( ), skewY( )

```
$displayItem->skewX(x)
$displayItem->skewY(y)
```

These methods add the given value to the current horizontal or vertical skew. See `skewXTo( )` and `skewYTo( )`.

skewXTo( ), skewYTo( )

```
$displayItem->skewXTo(x)
$displayItem->skewYTo(y)
```

These functions set the horizontal and vertical skew attributes for the item. The skew value is expressed as a real number where 0 indicates no skew and 1.0 is a 45 degree skew. Positive numbers indicate a counterclockwise skew anchored at the origin.

The SWF::Shape Module

The SWF::Shape object holds a data structure that represents a shape as described in Chapter 8. A shape consists of a series of points, a fill style, and a line style.

Example 9-2 uses the methods of the Shape object to draw a logarithmic spiral using the Golden Mean (see *http://mathworld.wolfram.com/GoldenRatio.html*).

The spiral starts at the origin and the pen moves in a counterclockwise direction. The direction of the curve is determined by cycling through the `@dx` and `@dy` arrays; the first segment should be drawn in the positive x and y directions, the second in the positive x, negative y directions, etc. The control points are always on the outside edges of the curve. The result is pictured in Figure 9-3.

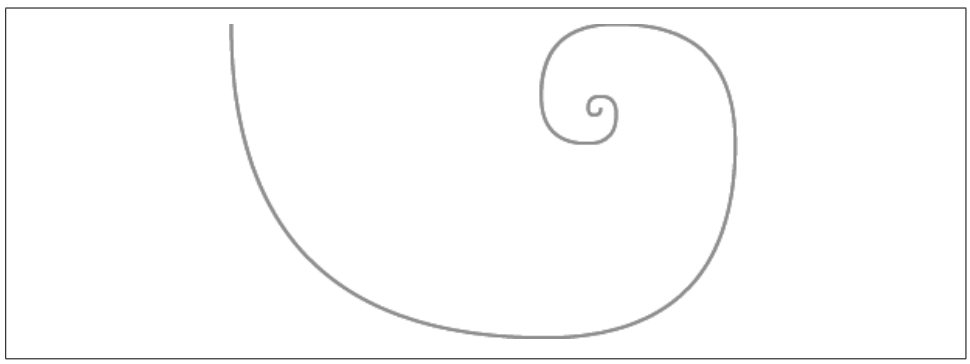


Figure 9-3. A logarithmic spiral shape

Example 9-2. A program for drawing a logarithmic spiral shape

```
#!/usr/bin/perl -w
#
# Example 9-2. An example using SWF::Shape( )

use strict;
use SWF::Shape;

SWF::setScale(1.0);
my $s = new SWF::Shape( );         # Create a new Shape object
$s->setLineStyle(1, 255, 0, 0);

my @dx = (1, 1, -1, -1);
my @dy = (1, -1, -1, 1);
my ($x, $y) = (0, 0);
my ($x1, $y1) = (0, 0);
my $w = 100;
my ($cx, $cy);

$s->movePenTo($x, $y);

for(my $i=0; $i <= 10; $i++) {

    $x1 = $x1 + $dx[$i%4] * $w;
    $y1 = $y1 + $dy[$i%4] * $w;

    if ($i % 2) {               # An odd turn
        ($cx, $cy) = ($x1, $y);
```

Example 9-2. A program for drawing a logarithmic spiral shape (continued)

```
    } else {                    # An even turn
        ($cx, $cy) = ($x, $y1);
    }

    $s->drawCurveTo($cx, $cy, $x1, $y1);  # Add a curve segment to the Shape
    $x = $x1;                             # Set the current point.
    $y = $y1;
    $w = $w * .618034;   # The width of the bounding box for the next curve segment
                         # is determined by the Golden Mean
}

# Create a Movie to hold the Shape

my $m = new SWF::Movie( );
$m->setDimension(300,300);
$m->add($s)->moveTo(0,10);
$m->nextFrame( );

$m->save("example9-2.swf");
```

You may notice that the curves in this shape seem a little flattened; this is because Ming (and the SWF specification) uses quadratic Bezier curves (with one control point) rather than the more flexible cubic Bezier curves (with two control points).

addFill( )

```
$fill = $shape->addFill(r, g, b [, a])
$fill = $shape->addFill(bitmap [, flags])
$fill = $shape->addFill(gradient [, flags])
```

Each shape contains a list of fill styles, any or all of which may be used to draw the shape. This method creates a new SWF::Fill object that can be used with `setLeftFill( )` or `setRightFill( )`. The method can be called in one of three ways:

1. If called with RGBA components, a solid fill is added to the shape's fill style list.
2. If called with an SWF::Bitmap object, a bitmap fill is added to the shape's fill style list. The *flags* argument can be:

   ```
   SWF::Fill::SWFFILL_TILED_BITMAP (the default)
   SWF::Fill::SWFFILL_CLIPPED_BITMAP
   ```

3. If an SWF::Gradient object is the first argument, a gradient fill is added to the shape's list of fill styles. The *flags* argument can be:

   ```
   SWF::Fill::SWFFILL_LINEAR_GRADIENT (the default)
   SWF::Fill::SWFFILL_RADIAL_GRADIENT
   ```

Note that the fill must be created with the `addFill( )` method, and each fill object is associated with a particular Shape object. In the case of a gradient fill, the gradient must be defined *before* the shape is created. Fill objects cannot be used interchangeably between shape objects. Figure 9-4 shows illustrations of the bitmap and gradient fill styles.

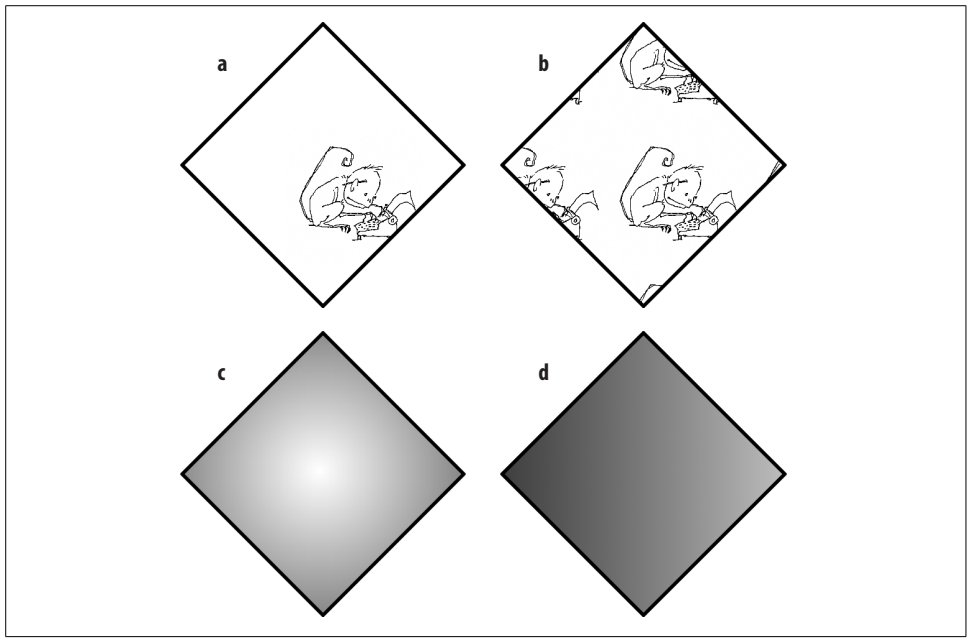


Figure 9-4. Bitmap and gradient fill flags: a) SWFFILL_CLIPPED_BITMAP, b) SWFFILL_ TILED_BITMAP, c) SWFFILL_RADIAL_GRADIENT, and d) SWFFILL_LINEAR_GRADIENT

drawCurveTo( )

```
$shape->drawCurveTo(controlX, controlY, x, y)
```

This method draws a curved line from the current point to (*x*, *y*) using (*controlX*, *controlY*) as a control point. After this operation, the current point is (*x*, *y*). Figure 9-5 illustrates a quadratic Bezier curve, where the curvature between two points is defined by a single control point. (Compare to the cubic Bezier curves used by PostScript in Chapter 10.)

drawCurve( )

```
$shape->drawCurve(controlDx, controlDy, dx, dy)
```

This method draws a curved line from the current point (*x*, *y*) to the point (*x+dx*, *y+dy*) using (*x+controlDx*, *y+controlDy*) as a control point. After this operation, the current point is (*x+dx*, *y+dy*).

drawCircle( )

```
$shape->drawCircle(r)
```

This method draws a circle with radius *r* centered at the current point. This operation does not affect the current point.

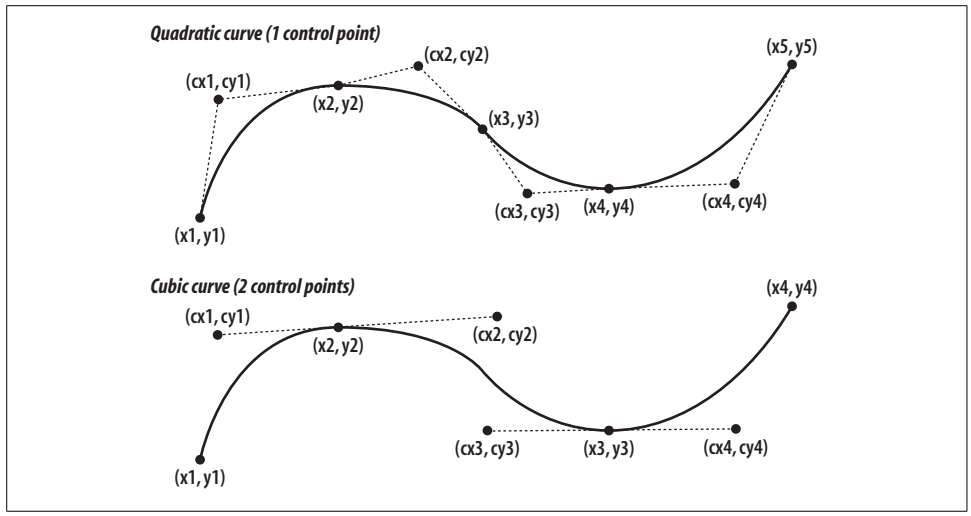


Figure 9-5. SWF uses quadratic curves (top) as opposed to cubic curves (bottom)

drawFontGlyph( )

```
$shape->drawFontGlyph(font, char)
```

This method draws a single character at the current point in the specified font, which must be an SWF::Font object. The origin of the baseline of the glyph is anchored at the current point.

drawLine( )

```
$shape->drawLine(dx, dy)
```

This method draws a straight line from the current point (*x*, *y*) to (*x+dx*, *y+dy*) using the current line style. After this operation, the current point is (*x+dx*, *y+dy*).

drawLineTo( )

```
$shape->drawLineTo(x, y)
```

This method draws a straight line from the current point to (*x*, *y*) using the current line style (which is set using `setLineStyle( )`). After this operation, the current point is (*x*, *y*).

movePenTo( )

```
$shape->movePenTo(x, y);
```

When drawing a shape, all coordinates are expressed in relation to the origin of the shape itself, not to the frame containing the shape. This method moves the current point of the pen to (*x*, *y*) in the shape's coordinate space.

movePen( )

```
$shape->movePen(dx, dy)
```

With the current point at (*x*, *y*), this method moves the pen to the point (*x+dx*, *y+dx*) without creating a path between the two points.

new( )

```
$shape = new SWF::Shape( )
```

This method returns a new Shape object.

setLeftFill( ), setRightFill( )

```
$shape->setLeftFill(fill)
$shape->setLeftFill(red, green, blue, [alpha])
$shape->setRightFill(fill)
$shape->setRightFill(red, green, blue, [alpha])
```

The SWF specification does not provide a way of defining shapes such that they have an "inside" area that can be filled with a particular fill style. Instead, each line in a shape has one fill style to its left and one fill style to its right. These fills can be set using the forms given above. In the first instance, *fill* must be an SWF::Fill object, which is created by the `SWF::Shape::addFill( )` method. For a simple colored fill, you can use the second form.

setLineStyle( )

```
$shape->setLineStyle(width, red, green, blue [, alpha])
```

This method sets the properties of the line used to stroke the shape. You provide the stroke width (in twips, subject to the scale defined by `setScale( )`) and color (with an optional alpha channel).

The SWF::Button Module

A Button is an object that can be used to track user input such as mouse movements, mouse clicks, or keyboard events. When defining a button, you must supply at least a hit area, which is a Shape object that describes the area of the button that is receptive to user input.

You may also supply Shape objects that are displayed when the button is in one of its three states:

Up
: Displayed when the button has not received an event.

Down
: Displayed when the mouse has been clicked on the hit area.

Over
: Displayed when the mouse cursor is moved over the button's hit area.

The SWF::Button module defines three methods and four alias methods.

addShape( )

```
$button->addShape(shape, state_flag)
```

This method adds a shape to one of the button's four states, indicated by the value of *state_flag*:

`SWF::Button::SWFBUTTON_HIT`
: The shape defines the hit area of the button. The shape is not displayed on the screen.

`SWF::Button::SWFBUTTON_UP`
: Displayed when the button has not yet been pressed.

`SWF::Button::SWFBUTTON_DOWN`
: Displayed when the button is pressed down on the hit area.

`SWF::Button::SWFBUTTON_OVER`
: Displayed when the mouse cursor is over the hit area of the button.

If a shape is not filled, only the path will act as a button, not the area bounded by the path. The next four methods can be used as aliases for setting the shape of each of the four states.

new( )

```
new SWF::Button( )
```

This method creates a new Button object.

setAction( )

```
$button->setAction(action, event_flag)
```

This adds an SWF::Action object to the button. The script is activated when the button receives one of the following events, as indicated by the *event_flag* parameter:

`SWF::Button::SWFBUTTON_MOUSEUP`
: Execute the script when the mouse button is pushed down over the button's hit area, then released. This is the default value for `$event_flag`.

`SWF::Button::SWFBUTTON_MOUSEOVER`
: Execute the script when the mouse cursor enters the button's hit area.

`SWF::Button::SWFBUTTON_MOUSEOUT`
: Execute the script when the mouse cursor leaves the button's hit area.

`SWF::Button::SWFBUTTON_MOUSEDOWN`
: Execute the script when the mouse button is pressed within the button's hit area.

`SWF::Button::SWFBUTTON_DRAGOVER`
: Execute the script when the mouse enters the button's hit area with the button pressed down.

SWF::Button::SWFBUTTON_DRAGOUT
: Execute the script when the mouse leaves the button's hit area with the button pressed down.

Consider the following snippet of code, where `$shape` is a previously defined SWF::Shape object, and `$movie` is a previously defined SWF::Movie:

```
my $b = new SWF::Button( );
$b->addShape($shape, SWF::Button::SWFBUTTON_HIT);
$b->addShape($shape, SWF::Button::SWFBUTTON_UP);
my $item = $movie->add($b);
$item->setName("button1");
$b->setAction(new SWF::Action("_root.button1._xscale++;"),
              SWF::Button::SWFBUTTON_MOUSEDOWN);
```

Every time the button is clicked, the button's horizontal scale is incremented.

setHit( )

`$button->setHit(`*shape*`)`

This method sets the shape that defines the hit area of the button. Same as:

```
$button->addShape($shape, SWF::Button::SWFBUTTON_HIT);
```

setDown( )

`$button->setDown(`*shape*`)`

This method sets the shape that is displayed when the mouse button is held while over the hit area. Same as:

```
$button->addShape($shape, SWF::Button::SWFBUTTON_DOWN);
```

setOver( )

`$button->setOver(`*shape*`)`

This method sets the shape that is displayed when the mouse cursor is over the hit area. Same as:

```
$button->addShape($shape, SWF::Button::SWFBUTTON_OVER);
```

setUp( )

`$button->setUp(`*shape*`)`

This method sets the shape that is displayed when the button is at rest (i.e., hasn't been hit). Same as:

```
$button->addShape($shape, SWF::Button::SWFBUTTON_UP);
```

The SWF::Bitmap Module

The SWF::Bitmap object can be used to add bitmap images read from external files. The image must be in JPEG or DBL format.

One of the design goals of Ming was that it not require any external libraries. As a result, you cannot directly read a PNG image into a Bitmap object; first you must convert it into the DBL format using the *png2dbl* tool provided in the *utils* directory of the Ming distribution. DBL is a Ming-specific format (okay, that's a euphemism for "hack") that stores an image file as a `DefineBitsLossless` block that can be easily incorporated into an SWF file. In the future it may be possible to read PNG files directly, so check the documentation for additional functionality in the latest version.

Example 9-3 shows how to incorporate a bitmap image into a movie by assigning it as a fill to a Shape object. The shape is used as a kind of clipping path for the bitmap, and can be used anywhere a shape can be used, for instance, as a button.

Example 9-3. Reading in a bitmap image

```
#!/usr/bin/perl -w
#
# Example 9-3. Reading in a bitmap image
#

use strict;
use SWF qw(Movie Shape Bitmap);

# Fill a Shape with a Bitmap. The .dbl file has been
# previously created from a PNG file using the png2dbl tool.

my $b = new SWF::Bitmap("bitmap.dbl");

# Fill must be created before the Shape is drawn

my $s = new SWF::Shape( );
my $f = $s->addFill($b);
$s->setRightFill($f);

# Get the dimensions of the bitmap and draw the Shape for the bounding box.
# A smaller bounding shape would act as a clipping path.

my $w = $b->getWidth( );
my $h = $b->getHeight( );
$s->drawLine($w, 0);
$s->drawLine(0, $h);
$s->drawLine(-$w, 0);
$s->drawLine(0, -$h);

my $m = new SWF::Movie( );
$m->setDimension($w, $h);
$m->add($s);

$m->save("example9-3.swf");
```

getHeight( )

```
$bitmap->getHeight( )
```

This method returns the height (in pixels, not twips) of the bitmap.

getWidth( )

```
$bitmap->getWidth( )
```

This method returns the width (in pixels, not twips) of the bitmap.

new( )

```
new SWF::Bitmap(file)
```

This method creates a new Bitmap object of the given filename. The file must be a JPEG or a DBL file, as described above.

The SWF::Text Module

The SWF::Text object can be used to draw simple blocks of colored text. The following code excerpt creates a two-line text block:

```
my $f = new SWF::Font("./Arial.fdb");
my $t = new SWF::Text( );
$t->setFont($f);
$t->setHeight(1200);
$t->moveTo(200,1200);
$t->setColor(255, 0, 0);
$t->addString("Foo ");
$t->moveTo(200, 2400);
$t->addString("Bar.");
```

Use the `SWF::Shape::drawGlyph( )` method if you want to draw characters with special fills or morph into other characters. If you need to create a block of text that can be updated dynamically, use the SWF::TextField object.

addString( )

```
$text->addString(string)
```

This method adds a string to the Text object at the current text cursor position.

getStringWidth( )

```
$width = $text->getStringWidth(string)
```

This method returns the width (in twips) of the given string as it is rendered using the Text object's current font and size settings. Note that the returned width takes into account the scale of the Text object.

moveTo( )

```
$text->moveTo($x, $y)
```

This method moves the position of the text cursor (i.e., the point at which text is added) to the specified point within the coordinate system of the Text object. The baseline of the text is drawn at the current text cursor point.

new( )

```
$text = new SWF::Text( )
```

This method creates a new Text object.

setColor( )

```
$text->setColor(red, green, blue [, alpha])
```

This method sets the text color; specify a red, green, blue, and optional alpha component.

setFont( )

```
$text->setFont(font)
```

This method sets the current font for the text object, where *font* is an SWF::Font object.

setHeight( )

```
$text->setHeight(height)
```

This method sets the font size, in twips (subject to any scaling in effect). The default height is 240.

setSpacing( )

```
$text->setSpacing(spacing)
```

This method defines the amount of space (in twips) between each character.

The SWF::TextField Module

Like the SWF::Text object, a TextField represents a block of text. However, TextFields cannot be transformed or skewed when displayed. The contents of a TextField can be changed over the course of a movie. (The Astral Trespassers example at the beginning of this chapter uses a TextField for the "Game Over" message.) TextFields are represented on the screen as user-editable text by default; this can be adjusted by creating the TextField with the appropriate flags.

addString( )

```
$textField->addString(string)
```

This method adds a string of text to the TextField, which is appended to the existing text.

align( )

```
$textField->align(alignment)
```

Unlike with SWF::Text objects, you can use a TextField to draw centered, justified, or left- or right-aligned text. Specify one of the following constants for the *alignment* parameter:

```
SWF::TextField::SWFTEXTFIELD_ALIGN_LEFT
SWF::TextField::SWFTEXTFIELD_ALIGN_RIGHT
SWF::TextField::SWFTEXTFIELD_ALIGN_CENTER
SWF::TextField::SWFTEXTFIELD_ALIGN_JUSTIFY
```

new( )

```
$textField = new SWF::TextField([flags])
```

This method creates a new TextField object whose behavior is determined by zero or more flags:

`SWF::TextField::SWFTEXTFIELD_NOEDIT`
: The TextField is non-editable.

`SWF::TextField::SWFTEXTFIELD_PASSWORD`
: User input is obscured by asterisks.

`SWF::TextField::SWFTEXTFIELD_DRAWBOX`
: A box is drawn bordering the TextField.

`SWF::TextField::SWFTEXTFIELD_MULTILINE`
: The TextField can accommodate multiple lines.

`SWF::TextField::SWFTEXTFIELD_WORDWRAP`
: On a multiline TextField, a string is wrapped to the next line once it reaches the margin of the field.

`SWF::TextField::SWFTEXTFIELD_NOSELECT`
: When the user clicks on the TextField, it is not selected.

More than one flag can be used by OR-ing them together:

```
use SWF::TextField qw(:TextField);
$textField = new SWF::TextField(SWFTEXTFIELD_MULTILINE |
                                SWFTEXTFIELD_WORDWRAP);
```

setBounds( )

```
$textField->setBounds(width, height)
```

This method sets the width and height (in twips, subject to scaling) of the bounding box of the TextField. Text in the field is cropped to this box. Note that you still have to set the font height with the `setHeight( )` method.

setColor( )

```
$textField->setColor(red, green, blue [, alpha])
```

This method sets the color of the text. The default is black.

setFont( )

```
$textField->setFont($font)
```

This method indicates the font that should be used to draw the text in the TextField. The parameter must be an SWF::Font object.

setHeight( )

```
$textField->setHeight(height)
```

This method sets the font size. The default value is 240 twips (subject to scaling), or 12 pixels.

setIndentation( )

```
$textField->setIndentation(width)
```

This method sets the indentation of the first line of the TextField.

setLineSpacing( )

```
$textField->setLineSpacing(height)
```

This method sets the amount of space between the bottom of the descender in one line and the top of the ascender in the next line. If you are familiar with typography, note that this is different from "leading." The default value is 40 twips, or 2 pixels.

setMargin( ), setLeftMargin( ), setRightMargin( )

```
$textField->setMargins(left, right)
$textField->setRightMargin(width)
$textField->setLeftMargin(width)
```

These methods allow you to set the margins of the text block (in twips, subject to scaling).

setName( )

```
$textField->setName(name)
```

This method assigns a label to the TextField, which can later be referenced with ActionScript to add or change the text in the TextField.

The SWF::Font Module

The Font object represents a set of glyphs that can be used to draw text. At this time, Ming does not directly support PostScript or TrueType fonts. The only type of font supported is a specially created FDB file.

An FDB file is a document containing a font encoded as an SWF Font Definition Block. To create an FDB file you must use the *makefdb* tool contained in the *util* directory of the Ming distribution. The *makefdb* program takes an SWT Generator template file and extracts the font information from it. To convert a particular font for use with Ming, follow these steps:

1. If you have Macromedia's Flash tool (Version 4 or 5), download the Generator Authoring Extensions, a set of free Flash plug-ins that can be used to make SWT files.
2. Add some text to a movie using the font that you wish to translate. Save this movie as an SWT file.
3. Run the *makefdb* program to extract the Font Definition Block from the SWT file.

As of this writing, there are no other solutions for generating FDB files without the Flash tool. You have to rely on the kindness of strangers, who have translated several popular fonts already (see *http://shawn.apocabilly.org/PFG/fonts*). The Generator Authoring Extensions used for creating SWT files are available as a Flash plug-in from *http://www.macromedia.com/software/flash/*.

new( )

```
$font = new SWF::Font($filename)
```

This method creates a new instance of a font from the given FDB filename.

The SWF::Fill Module

Each Fill object must be associated with a particular Shape object and cannot be used interchangeably with other Shapes. Because of this, there is no explicit constructor for a Fill. A new Fill object is returned by the `SWF::Shape::addFill( )` method. Fills can be one of three types: solid color (with or without an alpha channel), gradient, or bitmap. A gradient fill can be either linear or radial, and a bitmap fill can be tiled to fill the region, or clipped to fit the region.

The following constants are defined by this module:

```
SWFFILL_SOLID
SWFFILL_GRADIENT
SWFFILL_LINEAR_GRADIENT
SWFFILL_RADIAL_GRADIENT
SWFFILL_BITMAP
SWFFILL_TILED_BITMAP
SWFFILL_CLIPPED_BITMAP
```

The SWF::Fill module defines five methods that may be used to move the origin of the fill or to transform the fill. None of these methods affect solid fills.

moveTo( )

```
$fill->moveTo(x, y)
```

The origin of the fill is moved to the coordinate (*x*, *y*) in the coordinate space of the shape.

rotateTo( )

```
$fill->rotateTo(degrees)
```

This method rotates the fill from its original orientation.

scaleTo( )

```
$fill->scaleTo(x_scale, y_scale)
```

Each Fill object has a scaling attribute that is initially set to 1. The `scaleTo( )` function sets the horizontal and vertical scales to the specified values, replacing the current scale values.

skewXTo( ), skewYTo( )

```
$fill->skewXTo(x)
$fill->skewYTo(y)
```

These functions set the horizontal and vertical skew attributes for the Fill object. The skew values are expressed as real numbers where 0 indicates no skew, and 1.0 is a 45 degree skew. Positive numbers indicate a counterclockwise skew anchored at the origin of the fill.

The SWF::Gradient Module

A gradient consists of a list of color values, each value with a set position. The gradient can be scaled to fill a given region and can appear as a linear gradient or a radial gradient (illustrated back in Figure 9-4). The color of the points between the two defined end points gradually transforms from the color of the first point to the color of the second.

A gradient is constructed by adding color entries at particular positions:

```
my $gradient = new SWF::Gradient( );
$gradient->addEntry(0.0, 255, 0, 0);
$gradient->addEntry(1.0, 255,255,255);
my $fill = $s->addFill($gradient);
$fill->scaleTo(.1);            # A method of the SWF::Fill object
```

See "The SWF::Fill Module" for additional methods for controlling gradient fills.

addEntry( )

`$g->addEntry(`*ratio, red, green, blue* `[, `*alpha*`])`

This method is used to add color entries to a gradient object. The ratio is a number between 0 and 1 that represents the position of the color in the gradient. Calls to `addEntry( )` should be made in order of increasing ratios, or an error will occur.

new( )

`$g = new SWF::Gradient( )`

The `new( )` method creates a new, empty gradient object.

The SWF::Morph Module

A Morph is an object that encapsulates all the different transition states that represent the transformation of one shape into another. To use a Morph, you set an initial state and a final state (both SWF::Shape objects). As the first shape is transformed into the second, all of the original shape's attributes (color, rotation, etc.) are gradually adjusted to match those of the final state. Once the Morph is placed within a frame, you can set the state of the Morph to any of the infinite number of transitional states between the two extreme shapes using the `setRatio( )` method. A value of .5 displays an item that is halfway morphed between the starting and ending state. In the following example, the Morph changes from a square to a star in 10 frames, then changes back to a square in another 10 frames.

Both the initial and final shapes must contain an equal number of points. If the two states are defined by shapes with different numbers of points, you'll get inconsistent results in the final movie (typically random lines and noise).

In Example 9-4, a red square morphs into a blue eight-pointed star over the course of 10 frames, then morphs back into a square. Figure 9-6 shows a few of the stages.

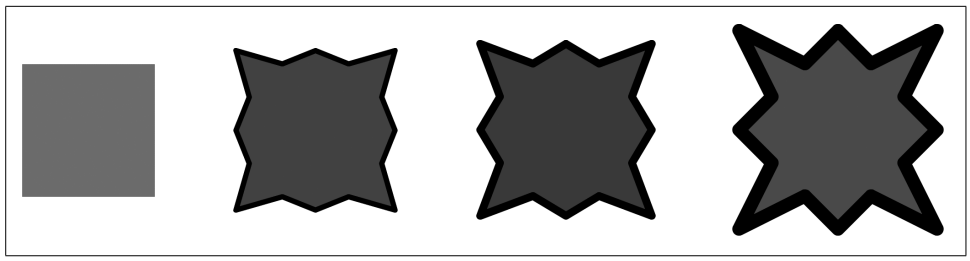


Figure 9-6. In a morph, the starting and ending shapes must have the same number of points

Example 9-4. Morphing between two shapes

```
#!/usr/bin/perl -w
#
# Example 9-4. Morphing between a square and a star.
#
use strict;
use SWF::Movie;
use SWF::Morph;
use SWF::Shape;
use SWF::DisplayItem;

SWF::setScale(1.0);

my $morph = new SWF::Morph( );
my $s = $morph->getShape1( );      # The initial state of the morph; a red square
$s->setLine(0,0,0,0);
$s->setLeftFill($s->addFill(255, 0, 0));
$s->movePenTo(0, 0);
$s->drawLine(500,0);   $s->drawLine(500,0);             # Top
$s->drawLine(500,0);   $s->drawLine(500,0);
$s->drawLine(0, 500);  $s->drawLine(0, 500);            # Right
$s->drawLine(0, 500);  $s->drawLine(0, 500);
$s->drawLine(-500,0);  $s->drawLine(-500,0);            # Bottom
$s->drawLine(-500,0);  $s->drawLine(-500,0);
$s->drawLine(0, -500); $s->drawLine(0, -500);           # Left
$s->drawLine(0, -500); $s->drawLine(0, -500);

my $s2 = $morph->getShape2( );  # The final state: a blue star with a thicker stroke
$s2->setLine(200,0,0,0);
$s2->setLeftFill($s2->addFill(0, 0, 255));
$s2->movePenTo(-500,-500);
$s2->drawLine(1000, 500);   $s2->drawLine(500, -500);   # Top
$s2->drawLine(500, 500);    $s2->drawLine(1000, -500);
```

Example 9-4. Morphing between two shapes (continued)

```
$s2->drawLine(-500, 1000);  $s2->drawLine(500, 500);    # Right
$s2->drawLine(-500, 500);   $s2->drawLine(500, 1000);
$s2->drawLine(-1000, -500); $s2->drawLine(-500, 500);   # Bottom
$s2->drawLine(-500, -500);  $s2->drawLine(-1000, 500);
$s2->drawLine(500, -1000);  $s2->drawLine(-500, -500);  # Left
$s2->drawLine(500, -500);   $s2->drawLine(-500, -1000);

my $m = new SWF::Movie( );
$m->setDimension(4000,4000);
$m->setBackground(0xff, 0xff, 0xff);

# Add the Morph object to the Movie

my $i = $m->add($morph);
$i->moveTo(750, 750);

for (my $r=0; $r<=20; ++$r)
{
   $i->setRatio(abs((10-$r)/10));
   $m->nextFrame( );
}

$m->save("example9-4.swf");
```

The SWF::Morph module defines three methods. The `setRatio( )` method belongs to the SWF::DisplayItem object; it can be used only after a Morph has been placed within a frame with the `add( )` method.

getShape1( )

```
$shape = $morph->getShape1( )
```

When the Morph is created, it creates two new SWF::Shape objects, one for the initial and one for the final state of the morph transition. This method returns a reference to the initial state Shape object, which should be used to draw the initial shape using any of the SWF::Shape functions (see also `getShape2( )`).

getShape2( )

```
$shape = $morph->getShape2( )
```

This method returns a reference to the final state Shape object of the Morph (see `getShape1( )`).

new( )

```
$morph = new SWF::Morph( )
```

This method creates a new Morph object.

The SWF::Sound Module

The SWF::Sound object allows you to read in an MP3 file that can be played in the background of a movie. Add the sound to a particular frame of the movie with the `SWF::Movie::setSoundStream( )` method, as in the following example:

```
my movie = new SWF::Movie;
my $sound = new SWF::Sound("goldenhair.mp3");   # A 30-second MP3
$movie->setRate(24);

# Make sure we have enough frames to accommodate the sound clip
# 30 seconds * 24 fps = 720 frames

$movie->setFrames(720);
$movie->setSoundStream($sound);
```

If the movie is shorter than the length of the sound clip, the sound is truncated, so we add enough frames to the movie to accommodate the entire 30 seconds.

new( )

`$sound = new SWF::Sound(`*`filename`*`)`

This creates a new Sound object. The *`filename`* should be the path to an MP3 file.

The SWF::Action Module

The SWF::Action object encapsulates a segment of ActionScript code that can be associated with a particular frame of a movie, a button event, or a particular DisplayItem (anything with an `AddAction( )` method). You write the ActionScript code and pass it to the ActionScript object as a string. When the SWF file is created, the JavaScript-like ActionScript is parsed and translated into ActionScript bytecode, which is embedded in the file. Ming uses different bytecode interpreters for Version 4- and Version 5-compatible code (as specified with the `SWF::setVersion( )` function).

new( )

`$a = new SWF::Action($script)`

The `new( )` constructor creates an Action object from a given string. The string should be a syntactically correct piece of ActionScript code. See Appendix D for a complete ActionScript reference.

SWF Recipes

As soon as you start using the Ming library to dynamically generate SWF files with Perl, you start to run into some issues that may make you think of turning to a tool like Flash. The simple act of drawing a circle, for example, is not as simple as you would think, as SWF only allows the use of cubic Bezier curves.

This section anticipates a few of these issues, and provides solutions to the following:

- Drawing a circle with cubic Bezier curves
- Creating a preloader with Ming
- Using a custom XML file format that can be used to assemble composite documents from previously created movie clips
- Using the XMLSocket ActionScript object to create a chat client that communicates with a Perl-based chat server

Drawing a Circle

The SWF specification allows you to represent curved lines using cubic Bezier curves. With only one control point, it is impossible to draw a perfect circle using only cubic Bezier curves. However, by applying a little math, we can get a pretty good approximation.

SWF generation tools such as Flash provide their own circle objects that construct a group of cubic Bezier curves to accurately represent a circle. Since we are creating an SWF file at a relatively low level (with the Ming library), we have to do some of the math ourselves. There's an upside to the hands-on approach, however—our circle generation script is able to draw a circle using any number of arc segments. This is useful for morphing with circles, where the starting shape must have the same number of points as the end shape.

Three points define a cubic Bezier curve: a start point, an end point, and a control point. The start and end points are easy to calculate. If we are drawing a circle of radius R made up of four quarter-circle segments, the starting and ending points would be (clockwise, starting at the origin in the SWF coordinate space):

Start angle	Start point	End angle	End point
0	(R, 0)	90	(0, R)
90	(0, R)	180	(-R, 0)
180	(-R, 0)	270	(0, -R)
270	(0, -R)	0	(R, 0)

Since we want to be able to divide the circle into any arbitrary number of curve segments (not just four), we can generalize this using the parametric equation for representing a circle:

$$x = R \cos(\theta)$$
$$y = R \sin(\theta)$$

where θ is the angle of the line (relative to the positive x axis) drawn from the origin to that point on the circle. The trigonometric functions in Perl use radians to represent angles instead of degrees, so we would actually use the values 0, $\pi/2$, π, and $3\pi/2$ for the angles.

Now that we know how to find the starting and ending points of each arc, we need to calculate the placement of the control point. We want the control point to be positioned such that the resulting curve follows the curvature of an ideal circle (as represented by our parametric equation above) as closely as possible. A cubic Bezier curve is represented by the following parametric equation:

$$x(t) = (1-t)^2 x_s + 2t(1-t)x_c + t^2 x_e$$
$$y(t) = (1-t)^2 y_s + 2t(1-t)y_c + t^2 y_e$$

where t is a number between 0 and 1. At the start of the curve, the x coordinate would be:

$$x(0) = x_s + 2*0(1-0)x_c + 0^2 x_e = x_s$$

At the endpoint, it would be:

$$x(1) = (0)^2 x_s + 2(1-1)x_c + 1^2 x_e = x_e$$

and at the midpoint:

$$x(.5) = (1-.5)^2 x_s + (1-.5)x_c + .5^2 x_e$$
$$= .25x_s + .5x_c + .25x_s$$

Since we want the Bezier curve to mirror the circle, we can say that the parametric equations for the circle and the cubic Bezier curve should be equal at the midpoint:

$$R \cos(\theta) = .25x_s + .5x_c + .25x_s$$
$$R \sin(\theta) = .25y_s + .5y_c + .25y_s$$

where θ is the angle of the midpoint of the curve. This is presented graphically in Figure 9-7, in which one segment, (a), is defined by the starting point (x1, y1), the ending point (x2, y2), and the control point (cx, cy). The angles $\theta 1$ and $\theta 2$ correspond to the starting and ending angles in Example 9-5, respectively.

With this bit of math figured out, we can write a routine `draw_arc()` that draws a circular arc segment. To draw a circle with an arbitrary number of curve segments, we simply need to iteratively call `draw_arc()` with the appropriate starting point, ending point, and midpoint angle, as in Example 9-5.

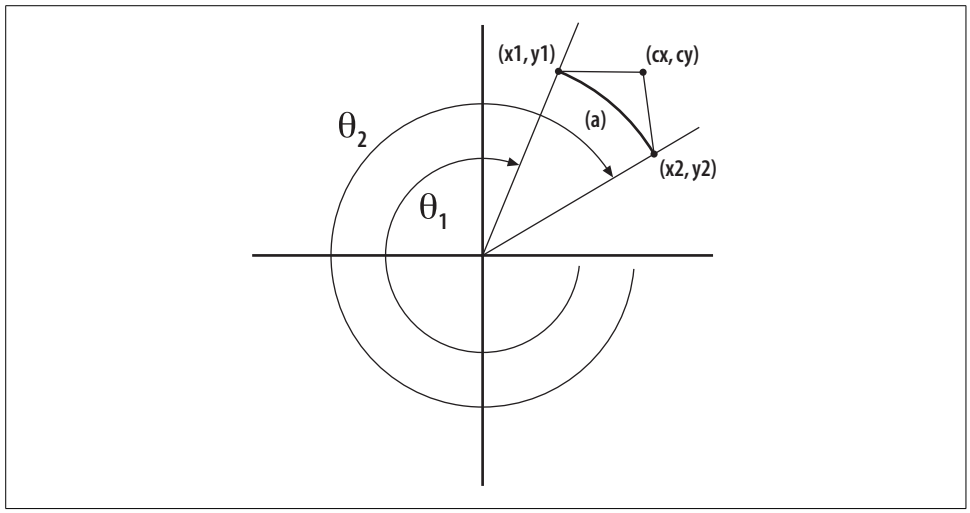


Figure 9-7. A circle can be approximated by a group of cubic Bezier curve segments

Example 9-5. Drawing a circle using an arbitrary number of cubic Bezier segments

```
#!/usr/bin/perl -w
#
# Example 9-5. Drawing a circle with cubic Bezier curves

use strict;
use SWF::Shape;

SWF::setScale(1.0);

my $PI = 3.1415926535;      # pi
my $r = 50;                 # the radius
my $start = 0;              # start angle
my $end = 0;                # end angle
my $segments = 6;           # number of segments

# Create a new Shape object

my $s = new SWF::Shape( );
$s->setLineStyle(1, 255, 0, 0);
$s->movePenTo($r,0);

for (my $end=360/$segments; $end<=360; $end+=360/$segments) {

    # Calculate the midpoint angle. The control point
    # lies on this line

    my $midpoint = $start+($end-$start)/2;

    # draw_arc( ) draws a circular arc between 2 points
```

Example 9-5. Drawing a circle using an arbitrary number of cubic Bezier segments (continued)

```
    draw_arc($s,
         $r*cos(radians($start)),$r*sin(radians($start)),
         $r*cos(radians($end)), $r*sin(radians($end)),
         radians($midpoint));
    $start=$end;
}

# Create a Movie to hold the Shape

my $m = new SWF::Movie( );
$m->setDimension(300, 300);
my $item = $m->add($s);
$item->moveTo(100,100);
$m->nextFrame( );

$m->save('circle.swf');

exit;

sub draw_arc {

    # Take a shape, a start coordinate, end coordinate and
    # pre-computed mid-point angle as arguments

    my ($s, $x1, $y1, $x2, $y2, $angle) = @_;
    my $cx = 2*($r*cos($angle)-.25*$x1-.25*$x2);
    my $cy = 2*($r*sin($angle)-.25*$y1-.25*$y2);

    # Draw the curve on the Shape

    $s->drawCurveTo($cx, $cy, $x2, $y2);
}

sub radians {
    return ($_[0]/180)*$PI;      # Convert to radians
}
```

The result is pictured in Figure 9-8. If you want to create a circle that morphs into a star with 15 points, simply set the `$segments` variable to 15. A circle made of 8 segments (i.e., eight 45 degree arcs) is the best-looking circle with the minimum number of points.

Creating a Preloader with Ming

Large SWF documents (say, over 100k) should display some sort of message telling the user to be patient while the document loads. The preloader has become an idiom of the Flash world (if not an art form); there's no good reason why you can't use one with Ming as well.

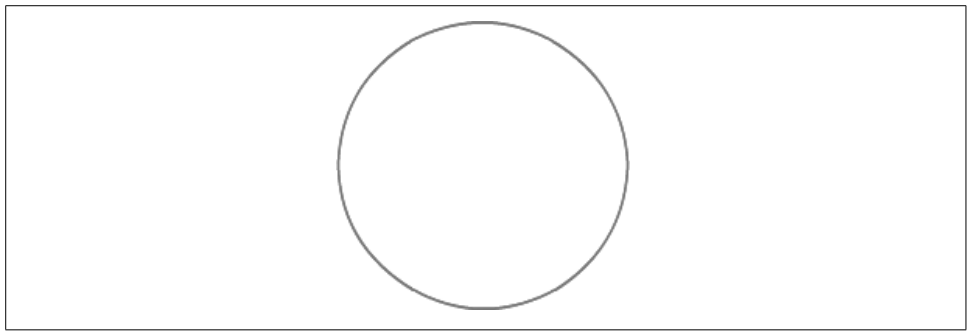


Figure 9-8. A near "perlfect" circle

The Flash plug-in starts playing your document as soon as it receives a complete frame. The function of a preloader is twofold: to give the user feedback, and to control the playback so that frames are not displayed before the whole movie is ready to be played.

A preloader is simply a couple of frames at the beginning of your document containing a couple of snippets of ActionScript. Sometimes people create preloaders that are complicated, artsy eye candy. This is fine, as long as it doesn't get so complicated that it requires its own preloader!

The preloader in this example is plain and functional: a simple red progress bar that displays the number of kilobytes loaded above it in a text field. The progress bar is a one-frame sprite that has ActionScript attached to it. The progress bar's script scales the sprite horizontally each time it loops, using the `getBytesLoaded()` and `getBytesTotal()` methods to calculate how much of the document has loaded so far.

The result of Example 9-6 is shown in Figure 9-9. When running this example, make sure that the bitmap is big enough that it takes some time to load; that is the whole point of a preloader, after all.

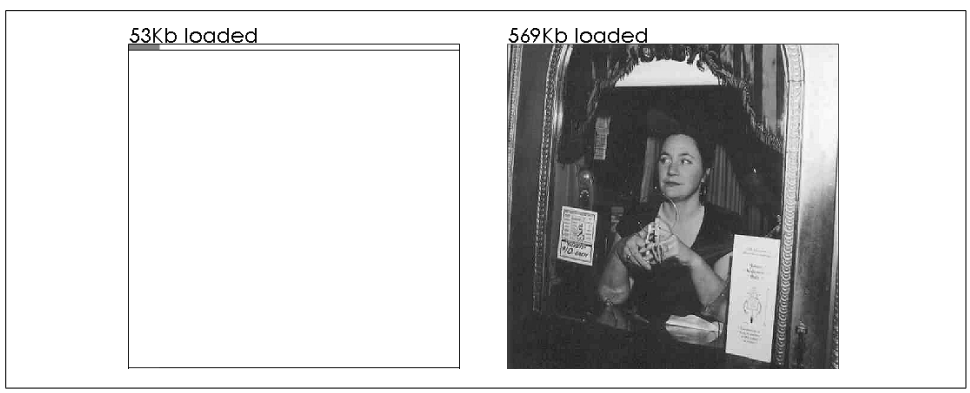


Figure 9-9. A preloader provides the user with an indication of the loading time of the document

Example 9-6. Creating an SWF file with a preloader

```
#!/usr/bin/perl -w
#
# Example 9-6.

use strict;

use SWF qw(:ALL);

SWF::useSWFVersion(5);

my $bitmap = new SWF::Bitmap("bitmap.dbl");
my $w = $bitmap->getWidth( );
my $h = $bitmap->getHeight( );

# Create a new Movie

my $m = new SWF::Movie;
$m->setDimension($w, $h+15);

# Create the feedback TextBlock. This displays a message: "X Kb loaded"

my $font = new SWF::Font("serif.fdb");
my $tf = new SWF::TextField( );
$tf->setFont($font);
$tf->setColor(0,0,0);
$tf->setName("feedback");
$tf->setHeight(10);
$tf->setBounds(300,50);
my $item = $m->add($tf);
$item->moveTo(0,2);

# Outline the bounding box of the bitmap

my $s = new SWF::Shape( );
$s->setLineStyle(1, 0, 0, 0);   # Black stroke
drawRect($s,$w,10,0,0);
$item = $m->add($s);
$item->moveTo(0,15);

# Create the preloader bar. It is a Sprite so that it can
# encapsulate its own ActionScript code for changing the
# scale based on the number of bytes loaded

my $s2 = new SWF::Shape( );
$s2->setLeftFill($s2->addFill(255, 0, 0));
drawRect($s2,$w,10,0,0);
my $bar = new SWF::Sprite( );
$item = $bar->add($s2);
$bar->nextFrame( );
$bar->add(new SWF::Action(<<ENDSCRIPT
    this._xscale = (_root.getBytesLoaded( ) / _root.getBytesTotal( )) * 100;
    _root.feedback = int(_root.getBytesLoaded( )/1024) + "Kb loaded";
```

Example 9-6. Creating an SWF file with a preloader (continued)

```
ENDSCRIPT
));
$bar->nextFrame( );
$item = $m->add($bar);
$item->moveTo(0,15);

my $s3 = new SWF::Shape( );       # Outline the bar in black
$s3->setLineStyle(1, 0, 0, 0);   # Black stroke
drawRect($s3, $w, $h, 0, 0);
$item = $m->add($s3);
$item->moveTo(0,15);
$m->nextFrame( );

# Create a loop with ActionScript. Jump out when the whole document is loaded.

$m->add(new SWF::Action(<<ENDSCRIPT
if (_root.getBytesLoaded( ) < _root.getBytesTotal( )){
   prevFrame( );
   play( );
} else {
   nextFrame( );
}
ENDSCRIPT
));
$m->nextFrame( );

# Add the bitmap

my $s4 = new SWF::Shape( );
my $f = $s4->addFill($bitmap);
$s4->setRightFill($f);
drawRect($s4, $w, $h, 0, 0);
$item = $m->add($s4);
$item->moveTo(0,15);
$m->nextFrame( );

$m->save("preloader.swf");

exit( );

# drawRect( ) is a helper procedure used to draw rectangles

sub drawRect {
    my $shape = shift;
    my ($w, $h, $dx, $dy) = @_;
    $shape->movePenTo($dx, $dy);
    $shape->drawLine($w, 0);
    $shape->drawLine(0, $h);
    $shape->drawLine(-$w, 0);
    $shape->drawLine(0, -$h);
}
```

Assembling Sprite-Based Documents with XML

One of the reasons to use a web application environment like ColdFusion is that it allows you to dynamically piece together documents that are composites of modular Flash files. Using Ming, it is relatively easy to roll your own XML format for describing a top-level document that is a collection of modular SWF files. This format could be used for integrating advertising with content dynamically, or for creating modular graphical user interfaces from pre-created components.

Let's say that you want to generate an SWF document that is built dynamically using a library of individual SWF movies. You could just piece these together in an HTML file using Cascading Style Sheets to position the movies on the page, but then the movies couldn't reference each other's Document Object Model, nor could you apply transformations. Another option is to create a simple XML format for representing a composite movie and writing an interpreter that translates the XML into a top-level SWF file.

We'll call the XML description format SWFscript. A sample SWFscript document would look like this:

```
<movie filename="index.swf" width="400" height="400">
    <sprite url="sprite2.swf">
        <scaleTo x="2" y="8"/>
        <moveTo x="100" y="100"/>
    </sprite>
    <sprite url="sprite2.swf">
        <scaleTo x="5" y="5"/>
        <moveTo x="200" y="200"/>
    </sprite>
    <sprite url="sprite2.swf">
        <moveTo x="300" y="300"/>
        <rotateTo degrees="45"/>
    </sprite>
    <nextFrame/>
</movie>
```

This document indicates that a top-level movie with the filename *index.swf* is created. The three modular SWF movies are read into the top-level movie as sprites and transformed according to the provided transformation tags.

The interpreter reads in the XML file and parses it using the XML::Parser module. Each time a start tag is encountered, the `start_tag( )` handler is called, and each time an end tag is encountered, the `end_tag( )` handler is called. The SWF file that is created is kind of a skeletal framework; each sprite is an empty SWF::Sprite that has a movie loader script attached to it. When the movie is loaded, each empty sprite uses the `loadMovie( )` method to replace itself (at its current position, subject to transformation) with the specified SWF file. The movie URLs are relative to the top-level document, so if you move the top-level movie, be sure to bundle the component files with it.

```
#!/usr/bin/perl -w
#
# Assemble a skeletal framework of Sprites

use strict;
use SWF qw(:ALL);
use XML::Parser;     # Used to parse the XML input file

my $filename = '';   # The output filename
my $sprites = 0;     # The number of <sprite> tags so far
my $m = undef;       # The top-level Movie
my $item = undef;    # The current DisplayItem

SWF::useSWFVersion(5);

my $parser = new XML::Parser(Handlers => { Start => \&start_tag,
                                           End   => \&end_tag,
                                         });
if (defined $ARGV[0]) {
    $parser->parsefile($ARGV[0]);  # Parse the input XML
} else {
    die "Please provide the name of an XML file.\n";
}
```

The start_tag( ) function is called whenever the parser encounters a start tag (<tag>):

```
sub start_tag {
    my $expat = shift;    # This is an Expat XML parser object
                          # that we don't use here.
    my $tag = shift;      # The tag name
    my %a = @_;           # The tag's attributes

    # Check the tag type

    if ($tag eq 'movie') {
        $m = new SWF::Movie;                  # Create a new Movie
        if ($a{'width'} && $a{'height'}) {
            $m->setDimension($a{'width'}, $a{'height'});
        }
    } elsif ($tag eq 'sprite') {
        if ($m) {
            $item = new_sprite($a{'url'});    # Create a new (empty) Sprite that
                                              # loads the external movie clip
        }
    } elsif ($tag eq 'moveTo') {
        if ($item) {
            $item->moveTo($a{'x'}, $a{'y'});  # Move the current item
        }
    } elsif ($tag eq 'scaleTo') {
        if ($item) {
        $item->scaleTo($a{'x'}, $a{'y'});     # Scale the current item
        }
    } elsif ($tag eq 'rotateTo') {
        if ($item) {
```

```
            $item->rotateTo($a{'degrees'});   # Rotate the current item
        }
    } elsif ($tag eq 'remove') {
        if ($item) {
            $item->remove( );                 # Remove the current item
            $item = undef;
        }
    } elsif ($tag eq 'nextFrame') {
        if ($m) {
            $m->nextFrame( );                 # Go to the next frame
        }
    }
}
```

The `end_tag( )` function is called when an end tag (`</tag>` or `/>`) is encountered. It takes two parameters: the XML::Parser object and the tag name. If the parser is at the end of a movie tag, output the SWF movie. Otherwise, discard the current sprite object.

```
sub end_tag {
    my ($expat, $tag) = @_;    # The Expat object and the tag name
    if ($tag eq 'movie') {
        $m->output( );
    } elsif ($tag eq 'sprite') {
    $item = undef;        # The sprite is no longer the current item
    }
}
```

The `new_sprite( )` function takes the URL of the external movie clip to be loaded and returns a new sprite with the ActionScript to load the movie clip at runtime.

```
sub new_sprite {
    my $url = shift;    # The URL of the external movie clip to load
    $sprites++;         # Increment number of sprites

    my $sprite = new SWF::Sprite( );
    $sprite->nextFrame( );
    my $i = $m->add($sprite);
    $i->setName("sprite$sprites");
    $m->add(new SWF::Action("loadMovie('$url','sprite$sprites');"));
    return $i;
}
```

Figure 9-10 shows a composite document created from three transformed copies of a component.

Communicating Between an SWF Client and a Perl Backend

SWF documents can easily communicate with other processes running on the Internet. Web and XML content may be loaded into an SWF file using ActionScript. In this section, we use ActionScript to create a simple chat client that can communicate with copies of itself via a central chat server written in Perl.

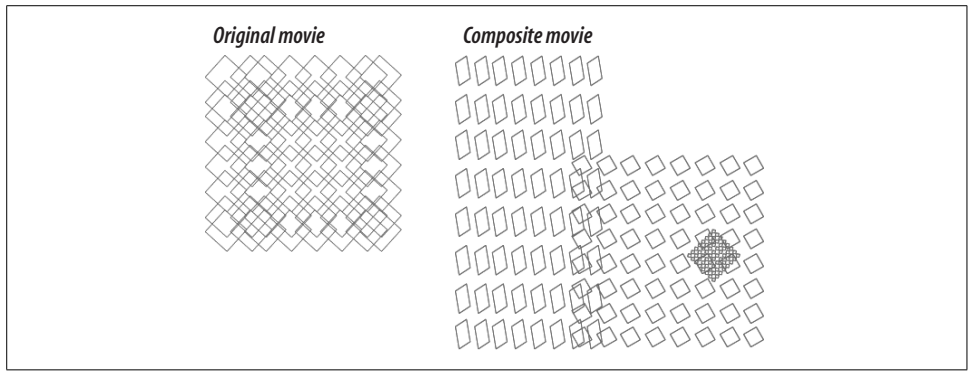


Figure 9-10. The composite on the right was created from an intermediate XML file format

First we'll write the simple chat server. It uses the IO::Select and IO::Socket modules (part of the standard Perl distribution) to open and manage the sockets for communicating to a number of clients. This server is a non-forking server; one instance of the server handles queued requests from many clients.

The IO::Socket::INET module (a subclass of IO::Socket) creates a new socket object that listens on a particular port (in this case, port 9999). The `Listen` attribute indicates how many connection requests are queued before new requests are refused (in this case, 10).

The IO::Select module provides a high-level interface to the system `select()` call. It provides the `can_read()` and `can_write()` methods that return a list of all the open sockets that have data waiting to be read from or written to.

The simple chat server looks something like Example 9-7. The client is shown in Example 9-8.

Example 9-7. A chat server that communicates with an SWF client

```
#!/usr/bin/perl -w
#
# Example 9-7. The server.

use IO::Socket;    # Part of the base Perl distribution,
use IO::Select;    # these are used for implementing socket
                   # communication

my $server = new IO::Socket::INET( LocalPort => 9999,
                                   Listen => 10);

my $select = IO::Select->new($server);
my @messages;      # The outgoing message buffer
$/="\0";           # Set the line input separator
print STDERR "Server started...\n";
```

Example 9-7. A chat server that communicates with an SWF client (continued)

```
# Loop in 'daemon mode' until killed

while (1) {
    @messages = ();

    # Loop through the open handles that are ready with data

    foreach my $client ($select->can_read(1)) {
        if ($client==$server) {

            # In this case, we have a new connection request
            $client = $server->accept();
            $select->add($client);
        } else {

            # Read the data from the client and push it on
            # the buffer, if the client is still there

            my $data = <$client>;
            if (defined($data)) {
                push @messages, $data;
            } else {
                $select->remove($client);
                $client->close;
            }
        }
    }

    # Loop through the handles waiting for input
    # and send them the buffered messages

    foreach my $client ($select->can_write(1)) {
        foreach my $m (@messages) {
            print $client "$m";
        }
    }
}
```

All this server does is:

1. Open a new connection for each connection request
2. Loop though the IO::Handle objects that have data waiting to be read
3. If it has received a connection request, open a new handle
4. Otherwise, read in the waiting data and push it into a buffer
5. Loop through all of the connections that are waiting to be written to and send them a copy of the data buffer

Up until now, we haven't been interested in the form of the input data because the server is simply forwarding the input data to the connected clients. Each client sends

chat message data to the server formatted as a single XML tag. Each tag is delimited by a null character (as required by the SWF specification), which explains why we set the data input delimiter to `"\0"` in the server.

ActionScript provides an easy way to send XML between applications with the XMLSocket object. This ActionScript object opens up a socket to a particular port on a given host computer and calls one of several callbacks when something interesting happens with the socket:

- `XMLSocket.onXML` is called when XML data is received.
- `XMLSocket.onConnect` is called when a connection is attempted.
- `XMLSocket.onClose` is called when a connection is closed.

These callbacks initially do not have any code associated with them. You must provide your own code for these hooks.

Our chat client, shown in Example 9-8, consists of three TextField objects and a submission button. The TextFields are for displaying messages, holding the user's name or "handle," and entering new messages, respectively. The first frame of the movie is a keyframe holding the ActionScript that sets up the environment and opens the socket to the chat server.

Example 9-8. An SWF chat client

```
#!/usr/bin/perl
#
# Example 9-8. The client.

use SWF qw(:ALL);
use SWF::TextField qw(:Text);

SWF::useSWFVersion(5);

# Create a new Movie

my $m = new SWF::Movie;
$m->setDimension(400,400);

# ActionScript for intializing the XMLSocket object that
# handles all of our communication and XML parsing needs

$m->add(new SWF::Action(<<ENDSCRIPT
    socket = new XMLSocket( );
    socket.connect('localhost', 9999);
    socket.onXML = dataReceived;
    socket.onConnect = welcome;
    socket.onClose = goodbye;

    function welcome (success) {
            if (success) {
                _root["chatwindow"] = "Welcome!\n";
```

Example 9-8. An SWF chat client (continued)

```
            } else {
                _root["chatwindow"] = "Couldn't connect!\n";
            }
        }

    function goodbye ( ) {
            _root["chatwindow"] = "Goodbye!";
        }

    function dataReceived (input) {
        var e = input.firstChild;
        _root["chatwindow"] = e.attributes.handle + ": " +
            e.attributes.text+"\n"+_root["chatwindow"];
        }
ENDSCRIPT
));
$m->nextFrame( );

# Create a TextField for displaying messages

my $font = new SWF::Font("serif.fdb");
my $tf = new SWF::TextField(SWFTEXTFIELD_DRAWBOX|
                            SWFTEXTFIELD_MULTILINE|
                            SWFTEXTFIELD_WORDWRAP);
$tf->setFont($font);
$tf->setColor(0,0,0);
$tf->setName("chatwindow");
$tf->setHeight(10);
$tf->setBounds(300,260);
my $item = $m->add($tf);
$item->moveTo(0,10);

# Create a TextField for holding the chatter's handle

my $tf2 = new SWF::TextField(SWFTEXTFIELD_DRAWBOX);
$tf2->setFont($font);
$tf2->setColor(0,0,0);
$tf2->setHeight(10);
$tf2->setName("handle");
$tf2->addString("Mr. Foo");
$tf2->setBounds(50,10);
$item = $m->add($tf2);
$item->moveTo(0,280);

# A TextField for entering new messages

my $tf3 = new SWF::TextField(SWFTEXTFIELD_DRAWBOX);
$tf3->setFont($font);
$tf3->setColor(0,0,0);
$tf3->setHeight(10);
$tf3->setName("message");
$tf3->setBounds(200,10);
```

Example 9-8. An SWF chat client (continued)

```
$item = $m->add($tf3);
$item->moveTo(50,280);

# Create a 'submit' button

my $s = new SWF::Shape( );
$s->setLineStyle(1, 0, 0, 0);   # Black stroke
$s->setLeftFill($s->addFill(255, 255, 255));
$s->drawLine(35, 0);
$s->drawLine(0, 10);
$s->drawLine(-35, 0);
$s->drawLine(0, -10);
my $t = new SWF::Text( );
$t->setFont($font);
$t->setColor(0, 0, 0);
$t->setHeight(10);
$t->addString("SUBMIT");
$item = $m->add($t);
$item->moveTo(260,290);

# Add the action script that sends the message and handle as
# XML to the chat server

my $b = new SWF::Button( );
$b->addShape($s, SWF::Button::SWFBUTTON_HIT);
$item = $m->add($b);
$item->setName("button1");
$b->setAction(new SWF::Action(<<ENDSCRIPT
    var xmlObj = new XML( );
    var m = xmlObj.createElement("message");
    m.attributes.handle = _root["handle"];
    m.attributes.text = _root["message"];
    xmlObj.appendChild(m);
    socket.send(xmlObj);
ENDSCRIPT
), SWF::Button::SWFBUTTON_MOUSEDOWN);
$item->moveTo(260,280);
$m->nextFrame( );

# Loop

$m->add(new SWF::Action(<<ENDSCRIPT
    prevFrame( );
    play( );
ENDSCRIPT
));
$m->nextFrame( );

$m->save('chatclient.swf');

exit( );
```

Figure 9-11 shows a sample session between two chatters.

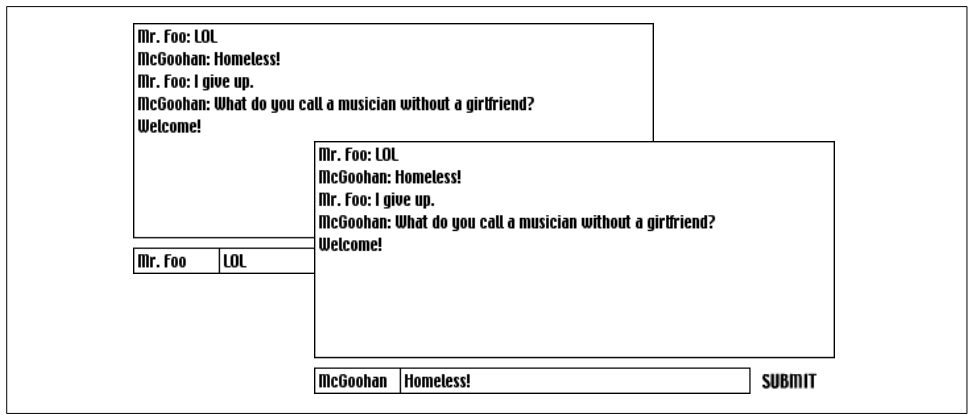


Figure 9-11. The ActionScript XMLSocket object allows you to communicate with other processes