Chapter 11

FLASH ANIMATION AND PUBLISHING YOUR GAMES

As we've seen in previous chapters, you can use a timer and some clever AS3.0 code to program your game objects with simple animations. With a bit of time and the right images, you can even make those animations really complex. Timer based animations might be all you need for your games, but for some very sophisticated animation or transition effects, you'll save yourself a lot of time and work by using some dedicated animation software. Flash Professional is the perfect tool for the job, and in this chapter I'm going to show you how to use it to create animations, import them into your game, and control them with AS3.0 code.

I'm also going to show you the steps you need to follow to publish your games for the web, desktop and mobile devices. Flash and AS3.0 are currently the only technologies that let you publish your games for all computer and mobile operating systems, and that's a powerful advantage you have as an AS3.0 developer to get your games into the hands of as many players as possible.

Animation with Flash Professional

Flash Professional is software made by Adobe that's specialized for creating animation for interactive Flash content. There was a time not so long ago that if you wanted to make Flash games, you could only do it with Flash Professional. You still can if you want – it has a really good interface for writing AS3.0 code, and it's easy to link your code with graphic objects that you draw on the stage. But as a game developer, you'll need more flexibility than it can give you, and this book has shown you how you can make Flash games using a whole handful of tools that you can pick and choose from, depending on which you think will do the job best.

But where Flash Professional still excels is as an excellent tool for creating 2D animation. In the first part of this chapter we'll look at some basic Flash animation techniques that you'll surely find some uses for in your games. I'll then show you how you can import those animations into your AS3.0 games and control them with code.

You'll find all the source code for this section in the folder called FlashProfessional in the chapter's source files. Open the file called start.fla if you want to work through the instructions in the next few section.

Importing artwork into Flash Professional

When you first start Flash Professional, select **`File > New`**. A **`New Document`** window will open asking you to select the kind of document you want to make, and the settings that the document should use. To keep your animation compatible with the projects in this books, select **`ActionScript 3.0`** as the project type and change the frame rate to **`60 fps`**. Set the **`width`** and **`height`** to whatever your stage's height and width is in your game project (It's been 550 by 400 for most of the projects in this book.) Click **`OK`** to create the document, and you'll see a new document open, as shown in Figure 11-1.

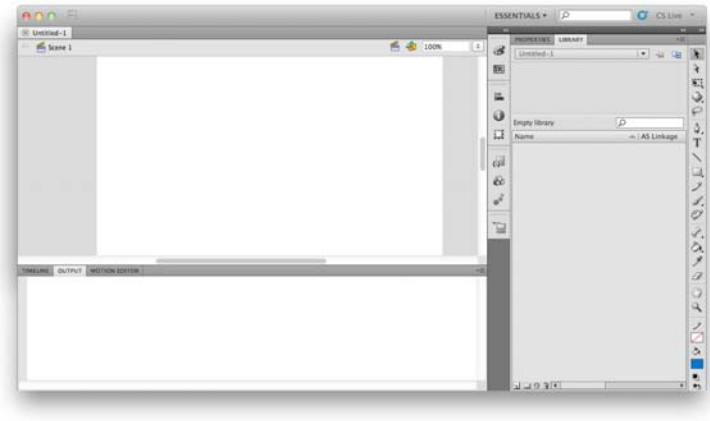

Figure 11-1. Flash Professional's workspace.

The blank drawing area near the top right is called the **stage**, and is where you'll be doing your animation. Flash Professional has some good drawing and illustration tools that, if you're used to using Photoshop or Illustrator, you'll find work in almost the same ways. For making artwork quickly that you can use in your animations, they're ideal. However, you may find you want to import some artwork that you've already created in Photoshop or Illustrator.

Flash Professional lets you import a huge variety of file types, and among those are Photoshop and Illustrator files. You can import Photoshop and Illustrator files and Flash Professional will keep them editable. If you've organized your work into layers, Flash Professional will give you option of importing the files with the layers intact, let you select layers to import, or let your import the file as a single, un-editable bitmap image.

To import an image file, select **`File > Import`** from the main menu, and then choose **`Import to Stage`** or **`Import to Library`**. If you choose **`Import to Stage`**, the imported image will appear on the main stage. If you choose **`Import to Library`**, it will first load the image into the **`Library`** panel, and you can drag it out onto the stage from there. Figure 11-2 shows what Flash Professional's Library panel looks like when I imported a PNG image of a rocket into it. The nice thing about importing an image into the Library is that, once it's there, you can make more copies of the same image simply by dragging another copy of it onto the stage.

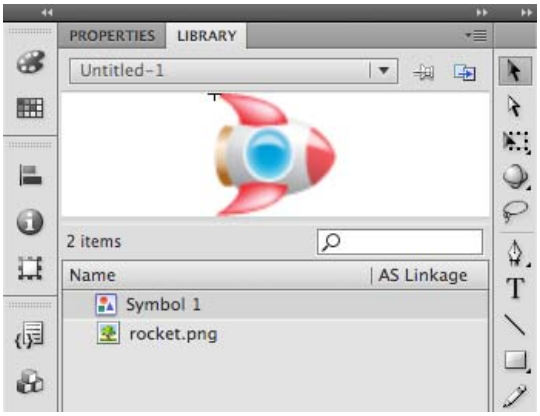

Figure 11-2. Import an image into the Library.

You can see from Figure 11-2 that when the rocket image was imported, another item called Symbol 1 was created. If you click on it, it looks exactly the same as the rocket.

The symbol is actually a class. It's a class that embeds the rocket.png image. Whenever you import an image into Flash Professional, it creates a class for it, and embeds the image into it. Flash Professional calls these classes *symbols*, but they're just like the plain old game object classes that we've been using to embed our images in most of the projects in this book. That means that the first thing you should do when you import the image is to change the symbol name to a useful class name. In this example, I renamed **`Symbol 1`** to **`Rocket`**, as you can see in Figure 11-3.

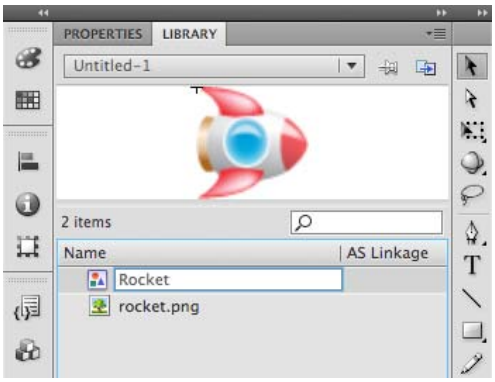

Figure 11-3. Symbols are classes that embed the imported image. Rename the symbol to a proper class name.

The animation that you do in Flash Professional will not be with the **`rocket.png`** image, but with the **`Rocket`** symbol.

Export for ActionScript

So if the **`Rocket`** symbol is actually a class, what kind of class is it? Click on the Properties tab and you'll see that it's a type of Bitmap, as shown in Figure 11-4.

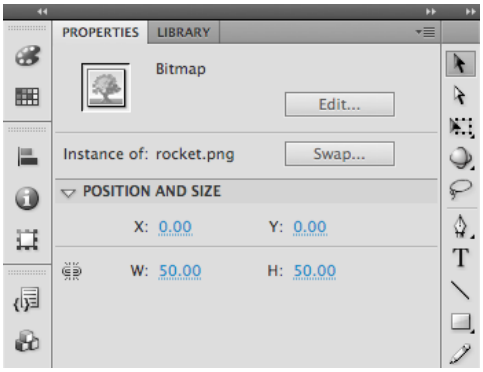

Figure 11-4. Imported PNG files are wrapped in a Bitmap class.

In chapter 1 I mentioned that display objects can be of three main types: Sprite, MovieClip or Bitmap. Flash Professional wraps all imported PNG files in a Bitmap class. This is actually fine, because it's all that we'll need for the simple animation we'll be doing in this chapter. But if you want a bit more flexibility change its type to MovieClip. MovieClip objects have the same properties as Sprite objects, but they also have a built-in timeline that you can access with AS3.0 code. You'll learn all about what the timeline is and how to use it for animation in the next section.

You can change a symbol from a Bitmap to a MovieClip by right-clicking the symbol's name in the **`Library`** and selecting **`Properties`**. The **`Symbol Properties`** window will open. Select **`MovieClip`** from the Type menu, Then click **`Export for ActionScript`** from the Linkage section. This is what will let you access this object with AS3.0 code in your game. Notice that the **`Class`** box is already filled in for you with the name of the symbol, which in this example is **`Rocket`**. It also tells you what the base class is: **`flash.display.MovieClip`**. This means that this new **`Rocket`** class will extend the **`MovieClip`** class. Figure 11-5 shows what the **`Symbol Properties`** looks like for the new **`Rocket`** object. (There are some additional options in this window, but you won't need to change any of them)

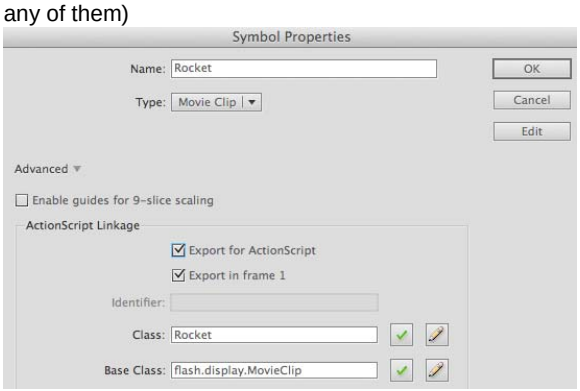

Figure 11-5. Export for Actionscript so that you can access and program the object with AS3.0 code.

Click **`OK`** when you've made these changes. You may see the following message window appear.

"A definition for this class could not be found in the classpath so one will be automatically generated in the SWF upon file export."

Select "don't show this again" and click **`OK`** to accept this. It just mean that Flash Professional is going to create the Rocket class for you when you publish the SWF file, and that's a good thing

Now that your symbol is a MovieClip object and it's been exported for ActionScript, you'll be able to access it in your program. But before I show you how to do that, let's take a closer look at what this new Rocket symbol is and how we can animate it with the timeline.

Creating animations

You have a choice when you're animating objects. You can either animate them directly on the main stage, or you can create animations inside the symbol itself. Both the main stage and the symbol have timelines that you use to do animation, and we'll look at how to work with both of them.

Working on the main stage

The main stage is the blank drawing space that you can see dominating most of the workspace. You can tell that you're working on the main stage because you'll see a **`Scene 1`** button just above and to the left of it. Look at the bottom of the main stage, and you'll see a panel called **`Timeline`**. Click on the **`Timeline`** panel and you'll see a row of numbered squares labeled Layer 1. Figure 11-6 shows what this looks like.

Figure 11-6. The main stage has a timeline which lets you animate objects.

Motion tween animation

We'll use the **`Timeline`** to animate the rocket on the main stage using a technique called **motion tweening**. It works by specifying the frame where the motion starts and then an ending frame where the motion stops. Flash calculates the position of the object for all the frames in between. That's where the *tween* comes from: it's short for *in-between.*

The first step is to create an *instance* of a symbol in the **`Library`**, and give it a name. I'm using the Rocket symbol in this example, but the steps are the same for any Movie Clip symbol, just change the names to symbol names you're using.

Drag an instance (a copy) of the **`Rocket`** symbol onto the main stage.

Make sure the new instance is selected; it will be surrounded by a blue bounding box if it is. Click the **`Properties`** tab, which you'll find to the right side of the main stage in the same panel set as the **`Library`**. The **`Properties`** panel is context sensitive, which means that the properties you can modify will change to match the kind of object you've selected.

Enter **`rocket`** in the instance name box. Figure 11-7 shows what your main stage and the **`Properties`** panel should now look like.

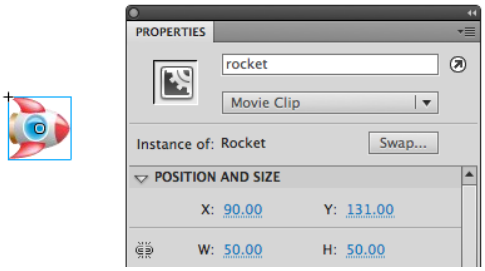

Figure 11-7. Create an instance of the Rocket symbol on the main stage.

When you drag an instance of a symbol onto the main stage like this, you're actually performing visually what you would write in code like this:

```
var rocket:Rocket = new Rocket();
```

The instance name that you give the object in the **Properties** panel is what you're going to use when you control it in a game with AS3.0 code. So for this reason it's important that any instance names you enter in the **Properties** panel match the same naming conventions for creating variables and objects that you would use in any AS3.0 program.

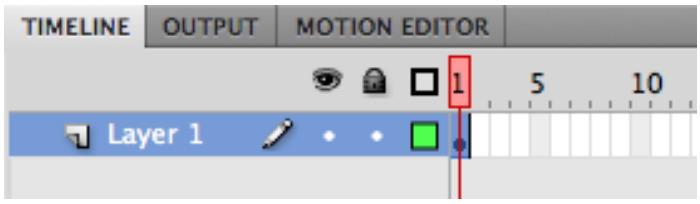

Figure 11-8. A black circle on the timeline is a keyframe, which represents a point where you can start an animation.

The **Timeline** represents all the frames in your animation. If your Flash movie is set to play at 60 fps, it means that your animation will play through 60 of these numbered frames each second. As soon as you've created an instance of the rocket on the main stage you'll notice that a black circle appears on frame one of the **Timeline**, as shown in Figure 11-8. A black circle indicates that you have a graphic on that frame that you can animate. It's called a keyframe, and represents a point at which an animation starts or stops. Empty circles are keyframes that don't yet contain any graphics.

> *Flash Professional uses layers, just like Photoshop and Illustrator. You can use them in the same way for layered animation effects.*

When you have a black keyframe on the **Timeline**, you can start animating. Let's animate the **rocket** instance with a simple motion tween.

1. Select the black keyframe in Frame 1 of Layer 1 in the **Timeline**. (If the Timeline is not visible, select Window ➢ Timeline.)
2. Select Insert ➢Motion Tween. You'll notice that the frames extend to frame 60 and are colored blue, as seen in Figure 11-9. This represents the duration that you want the animation to last

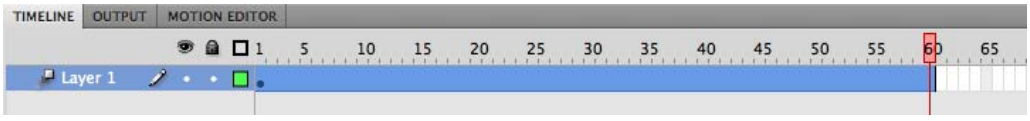

Figure 11-9. Insert a motion tween.

3. Drag the blue bar to frame 120.
4. Select frame 60.
5. Move the rocket to a new position on the stage. You'll notice two things:

- A tiny diamond- shaped dot appears in the timeline. This is called a **property keyframe**. Property keyframes are a special type of keyframe that indicate that the x or y position of an

object in an animation has changed. Unlike ordinary keyframes, they don't create new instances of the object. You can add property keyframes simply by highlighting the frame you want your object to move to and dragging the object there with the mouse.

- A **motion path** appears on the stage, which shows the path that the object has taken. You can actually select this path to bring up the motion path Properties panel, which allows you to apply some extra options such as rotation or **easing**. (Easing refers to how gently the object "eases into" its movement. Play around with it—it's fun!) You can also change the shape of the motion path so that the object takes a curved route to its destination. You can select a motion path to delete it as well, and when you do so, its associated property keyframe also disappears.

Figure 11-10 shows what the rocket keyframe and motion path look like.

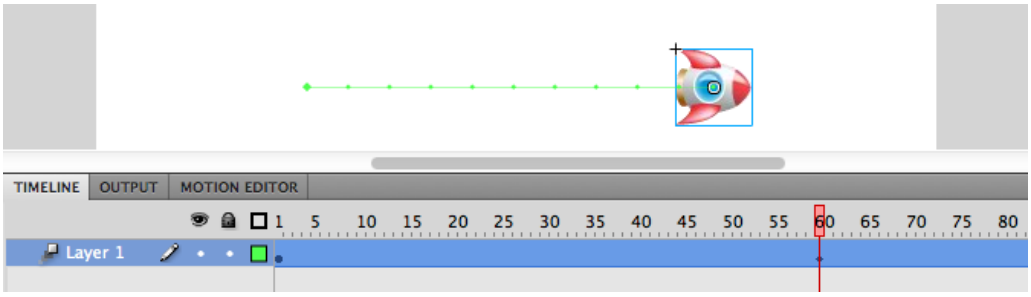

Figure 11-10. The property keyframe and motion path

6. The last step is to select frame 120 and move the rocket back to its start position.
7. To see what the animation looks like, press the Enter key or select **`Control > Play`** from the main menu. You can also press the play button at the bottom of the workspace. You'll see the rocket move back and forth along the motion path.

These are the basics of timeline animation, but there's much more you can do.

Apply some easing

You can control easing along the motion path to make the rocket speed up and slow down in a more natural way.

Select the path, and opening the **`Motion Properties`** panel.

Enter **`100`** in the **`Ease`** input field.

Play the animation again, and you'll see that the rocket gradually slows to a stop when it returns to its start

A value of 100 makes the easing effect very pronounced. For a subtler effect, try numbers less than 100 like 75 or 50. If you want the easing effect to start slowly and end quickly, enter a negative number, like -100.

You can modify and customize the easing effect using the Motion Editor.

Select the motion path and open the **`Motion Editor`** panel.

Scroll to the bottom of the panel to the **`Eases`** section. Click the **`add`** button and select Bounce from the option list, as shown in Figure 11-12. This will add bounce style easing as an option that you can apply to your animation. You'll see the number 4 next to the Bounce effect when you add this; this

determines the number of times the object will appear to bounce. You can change it to any number you like.

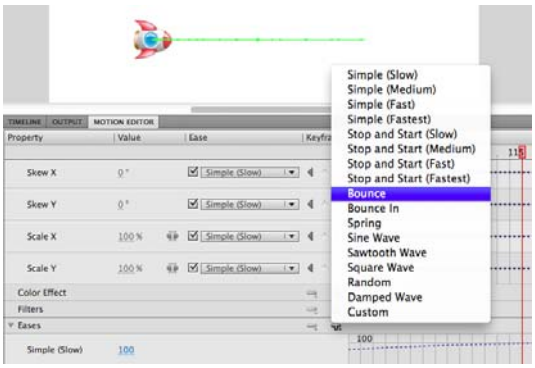

Figure 11-12. Add a Bounce easing effect.

The next step is to actually apply the Bounce easing effect to the motion path. Select Bounce from the Basic Motion option list, as shown in Figure 11-13.

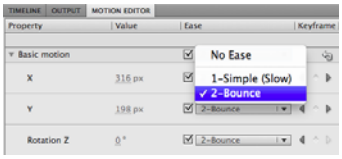

Figure 11-13. Add Bounce style easing to the motion path.

Play the animation again, and you'll see that the rocket now gradually bounces to a stop.

If you double-click on the Bounce effect from the Eases section, it expands to show you a graph of what the easing effect looks like over the span of your motion path, as shown in Figure 11-14.

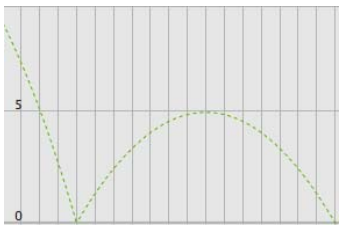

Figure 11-14. A graph describes the easing effect over the span of the motion path.

If you double-click on the X option of the Basic section, you'll see how the graph is overlaid onto your motion path, shown in Figure 11-15.

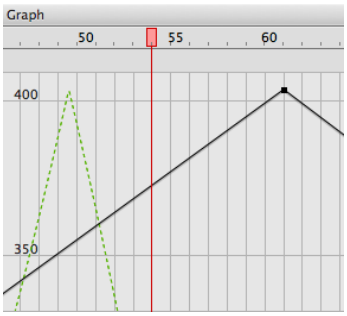

Figure 11-15. When you apply an easing effect, a graph shows you how the effect changes the position of your object over the span of the motion path.

> *This graph isn't editable, but you can make a custom ease that is. In the* ***Eases*** *section, click the* ***add*** *button and select* ***custom*** *from the option menu. This will add a new ease called custom, and you'll be able to change the shape of the graph using Bezier path editing tools; the same way you change the shape of a path using the Pen tool. You'll probably find that you'll never need to create a custom ease for games, as presets cover all the basics. but if you do, Adobe publishes a detailed animation guide that explains exactly how to do this: http://www.adobe.com/devnet/flash/learning_guide/animation/part09.html#adding_preset*

Saving your work and creating an SWF file

To be able to use this animation in an AS3.0 program, first save your work, and then export it as an SWF file.

Select **`File > Save As`** and give your file the name **`mainStageAnimation`**. Find a suitable place on your computer to save this file, and click the **`Save`** button. Flash Professional files are saved with the extension ".fla". The next step is to create an SWF file that we can use in an AS3.0 program.

Select **`File > Publish Settings`**. Un-check **`HTML Wrapper`** in the Other Formats section. This prevents Flash Professional from generating an HTML page for the SWF file. You can change the SWF file's name or any other properties form this option window.

Select **`File > Publish`**. Flash Professional now creates an SWF file for you called **`mainStageAnimation.swf`** in the same location that you chose to save the FLA file.

This new SWF file is the one that you're going to embed and control in a simple AS3.0 program in the next section.

Using AS3.0 to control an SWF animation

In the chapter's source files you'll find a simple example of how to load an SWF into a program and control the animation on the main timeline using AS3.0 code. Open the **`AnimationControl`** project folder and run the SWF. You'll see the rocket on the main stage, in exactly the same position it was placed on the stage in Flash Professional. A trace message tells you the animation's current frame (1) and its total frames (120). Click anywhere on the stage and the rocket's animation will start. SWF movies loop when they reach the end of the animation on the main timeline, so the rocket moves back and forth continuously across the stage. Figure 11-15 shows what you'll see.

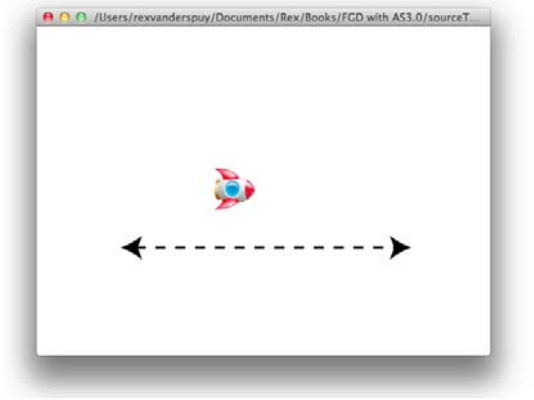

Figure 11-16. Click the stage to make the rocket animation start.

It's possible to embed an SWF file using the [Embed] metatag, just like we've been embedding sounds and images. I'll you how to do this later in chapter. But for the next few examples we'll look at how to load the SWF using the **URLRequest** and **Loader** class, in exactly the same way we loaded images into our programs in Chapters 3. You'll need to know how to do this at some point in your game design career. I'll also show you how to create a listener so that the SWF is only added to the stage after it's finished loading. This is important so that code doesn't try and access methods or properties on the loaded SWF file before its finished loading, which could result in errors.

Here's the complete **AnimationControl** application class that does all this work:

```
package
{
import flash.display.Loader;
import flash.display.MovieClip;
import flash.display.Sprite;
import flash.events.Event;
import flash.events.MouseEvent;
import flash.net.URLRequest;

  [SWF(width="550", height="400",
backgroundColor="#FFFFFF", frameRate="60")]

public class AnimationControl extends Sprite
{
//Create the objects needed to load and display the SWF
private var _swfURL:URLRequest
      = newURLRequest("../swfs/mainStageAnimation.swf");
private var _swfLoader:Loader = new Loader();
private var _animation:MovieClip;

public function AnimationControl()
    {
//Load the SWF and add a listener that runs when
       //the file is loaded
      _swfLoader.load(_swfURL);
        _swfLoader.contentLoaderInfo.addEventListener
        (Event.COMPLETE, swfLoadHandler);
```

```
//An mouse event listener starts the animation
      //when the stage is clicked
stage.addEventListener
(MouseEvent.MOUSE_DOWN, mouseDownHandler);
    }
private function swfLoadHandler(event:Event):void
    {
       //Copy the event handler's currentTarget.content property
       //into the _animation MovieClip object
      _animation = event.currentTarget.content;

//Add the _animation to the stage
stage.addChild(_animation);

//Stop the animation at the first frame
     _animation.stop();

//Display the animation's total frames and current frame
trace(_animation.currentFrame);  //Displays 1
trace(_animation.totalFrames);  //Displays 120
    }
private function mouseDownHandler(event:MouseEvent):void
    {
//Play the animation when the stage is clicked
     _animation.play();
    }
  }
}
```

The program first needs to import the MovieClip class, which is the type of object that the loaded SWF need to be.

```
importflash.display.MovieClip;
```

MovieClip objects are identical to **Sprite** objects except that they include a timeline as well as additional properties and methods you can use to control that timeline.

The SWF is in the project folder in a sub-folder called **swfs**. An **URLRequest** object is created that points to the file location, and a **Loader** object is created that will do the job of loading the file.

```
private var _swfURL:URLRequest
  = newURLRequest("../swfs/mainStageAnimation.swf");
private var _swfLoader:Loader = new Loader();
```

The loaded SWF is finally going to be stored in a MovieClip object called **_animation**, which is created in the class definition.

```
private var _animation:MovieClip;
```

When the constructor method runs, it uses the loader to load the SWF from its file location. It also adds a **COMPLETE** event listener to it.

```
_swfLoader.load(_swfURL);
_swfLoader.contentLoaderInfo.addEventListener
  (Event.COMPLETE, swfLoadHandler);
```

COMPLETE listeners are triggered when a file is finished loading. That means the **swfLoadHandler** is called when the loader has finished loading the SWF file. The first thing it does is copy the contents of its **event.currentTarget.content** property into the **_animation** object.

```
private function swfLoadHandler(event:Event):void
{
  _animation = event.currentTarget.content;
```

The **event.currentTarget.content** property is what contains the loaded SWF file. Once it's in the **_animation** object, it's easy to control.

The animation that we created in Flash Professional was set to start playing right away when the movie starts. We want to prevent that, so that it only starts when the user clicks the stage. A method called **stop** stops an animation from playing, at whatever position it's at in the timeline. After it's added to the stage, the **_animation** object's **stop** method is called to prevent it from playing right away.

```
stage.addChild(_animation);
_animation.stop();
```

MovieClip objects have a property called **currentFrame** that tells you the current position of the playhead on the timeline. When an SWF is first loaded, the playhead will be at frame one. The next line of code confirms this by displaying the value of **currentFrame** as a trace message.

```
trace(_animation.currentFrame);
```

It displays 1.

Another MovieClip property called **totalFrames** tells you the total number of frames in the animation, and another trace message displays this for us.

```
trace(_animation.totalFrames);
```

This displays as 120, which you'll recall is exactly the number of frames in the animation we created in Flash Professional. You can't change the values of either the **currentFrame** or **totalFrames** properties, but they're useful for finding out what your animation is currently doing.

When the user clicks the stage, the **mouseDownHandler** calls the SWF's **play** method, which starts the animation.

```
private function mouseDownHandler(event:MouseEvent):void
{
        _animation.play();
}
```

If you need to tell the SWF's timeline to go to a specific frame, you can use the **gotoAndPlay** or **gotoAndStop** methods. You use them like this:

```
gotoAndPlay(23)
gotoAndStop(23)
```

In both cases, this will make the playhead jump to frame number 23, and either play from or stop at that position. You can see this effect in the **AnimationControl** example by changing the code in the **mouseDownHandler** to the following code in bold:

```
private function mouseDownHandler(event:MouseEvent):void
```

```
{
        _animation.gotoAndPlay(1);
}
```

Now, whenever the stage is clicked the rocket animation will jump back to the first frame and start playing from the beginning again.

Table 11-1 describes how to use these all these MovieClip methods and properties to control animations.

Table 11-1. Methods and properties of Movie Clips animated on the timeline

Method	What it does
Play()	Starts the animation.
Stop()	Stops the animation.
gotoAndPlay()	Tells the object to move to a specific frame and then play from that point forward. For example, gotoAndPlay (23) moves the timeline's playhead to frame number 23 and starts playing from there.
gotoAndStop()	Stops the animation at a specific frame. gotoAndStop (23) stop the animation at frame 23.

Property	What it does
CurrentFrame	Tells you the number of the frame where your animated object is.
totalFrames	Gives you a number that is the total number of frames in your animation.

It's possible to add ActionScript code directly to the timeline in Flash Professional. For example, you could add a stop method to frame one of the animation to prevent it from playing right away.

To do this, create a new layer in the Timeline called Actions. Click on the empty keyframe at frame one, or add a new keyframe on the frame where you want the animation to stop. To add a new keyframe, right-click on the frame and choose Insert keyframe from the option menu. Make sure the keyframe is selected, and open the Actions panel, by selecting Window > Actions from the main menu. The Actions panel allows you to enter any AS3.0 code directly on the timeline or on an object. Add the following to the Actions panel:

stop();

You'll see a little lowercase "a" appear above the keyframe on the new layer where you added this action that confirms this frame contains code. Now when you load the animation into your AS3.0 program, it won't start playing right away.

Although you can add any AS3.0 code you like to any frame or object in Flash Professional, I don't recommend you do this, except perhaps for this type of very basic control over timeline animations. The reason is because your code will quickly become very difficult to maintain if it's scattered across multiple frames and multiple applications. Try and keep all of your code in your IDE (Flash Builder or Flash Develop) so that it's easy to find and change.

You may recall that when we created the animation in Flash Professional, we gave the rocket an instance name: **rocket**. You can access this rocket instance with AS3.0 code like this:

```
_animation.rocket
```

You can now change all of its properties, like **height**, **width**, **alpha**, **visible** or any of the other properties we've used in this book. MovieClip objects have the same properties as Sprite objects.

Orienting to a motion path

The motion path your animated object follows doesn't have to be in a straight line. It's very easy to create a path that follows an irregular or even circular path, and have your object follow that path in perfect alignment. The steps below will show you how.

Return to Flash Professional and select and delete any previous motion path you were using to animate the rocket.

Select frame 30, and move the rocket to a new location on the stage, as shown in Figure 11-17. A new property keyframe (black diamond) will be added on the timeline at the point, and the motion path will show you the path the animation will take.

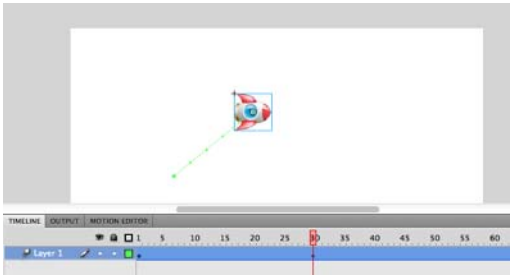

Figure 11-17. Move the rocket to a new location on frame 30.

Select frame 60 and move the rocket to another new location, as shown in Figure 11-18.

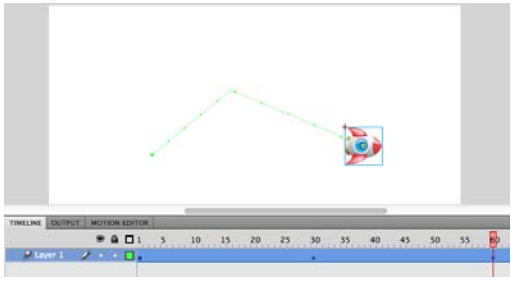

Figure 11-18. Move the rocket to another new location at frame 60.

Move the rocket to a new location at frame 90, and move it back to its starting point at frame 120. You'll end up with a motion path that looks a bit like Figure 11-20.

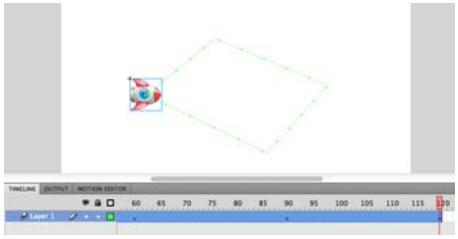

Figure 11-20. Move the rocket to new locations at frames 90 and 120 to bring it back to its start point.

You could leave the motion path like this, but let's curve it using the Selection Tool (the black arrow.)

Choose the Selection tool from the Tools panel and use it to bend the lines that make up the motion path. See if you can end up with a path that's smoothly curved, as shown in Figure 11-20.

Choose the Free Transform tool and rotate the rocket so that it's nose is aligned with the path, as shown in Figure 11-21.

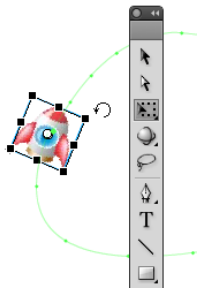

Figure 11-21. Rotate the rocket so that it's aligned with the path.

Click the motion path with the **`Selection`** tool (black arrow) to select it. Open the **`Properties`** panel and select the **`Orient to Path`** option, as shown in Figure 11-22. This option will keep the rocket's nose aligned to the path when its animated.

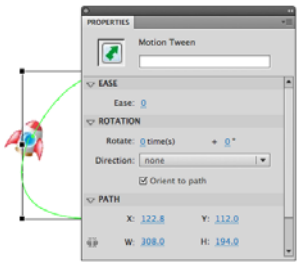

Figure 11-22. Select Orient to Path in the motion path Properties panel.

Play the animation, and you'll see that the front of the rocket stays perfectly aligned with the path, as you can see in Figure 11-23.

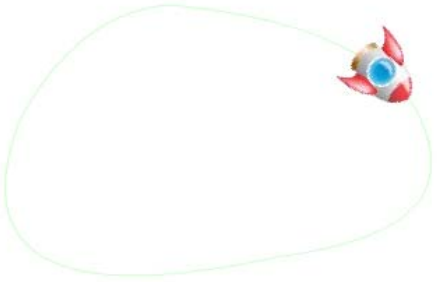

Figure 11-23. The rocket stays oriented to the path when the animation is played.

> *You can also use the motion path Properties panel to make an object rotate while it's moving along the path. You can make it rotate clockwise or counter clockwise, and also set the number of times it should rotate over the course of its journey across the path.*

Working with sub-objects

If you double click on the rocket, you'll see that the main stage dims and the word **`Rocket`** appears next to a Movie Clip icon at the top of the stage. You'll also notice that the main timeline is replaced with a new timeline, consisting of only one frame. This is the **`Rocket`** symbol's own timeline, and the stage is now the stage *inside* the Rocket symbol. Click on **`Scene 1`** at the top of the stage or double-click anywhere inside the rocket's stage to return back to the main stage.

Symbols have their own internal stages and timelines. You can build complex animated effects by using these internal timelines to animate objects inside other objects. Let's look at a really simple example of how this works by adding a flame to the rocket.

Creating a new symbol

We'll first create a new symbol for the flame animation, and then add it to the rocket.

8. Select **`Insert > New Symbol`** from Flash Professional's main menu.
9. Give it the symbol name **`Flame`**, make sure its type is **`Movie Clip`**, and select **`Export for ActionScript`**. Then click the **`OK`** button.

A new workspace will open. The word **`Flame`** will appear next to a Movie Clip icon at the top of the stage, and you'll have a new, blank timeline with one empty keyframe. Open the **`Library`** panel and you'll see that the **`Flame`** symbol is now a part of the **`Library`**. You're now working inside the Flame symbol, on its own internal stage.

10. Use Flash Professional's drawing tools to draw a simple orange flame shape on the stage. The black crosshair tells you where the center of the stage is, so try to align your drawing under it. When you're finished your drawing, you'll see that the keyframe on frame one in the timeline is now black. That shows you that there's a graphic on that frame. Figure 11-24 shows what the first frame of my Flame symbol looks like.

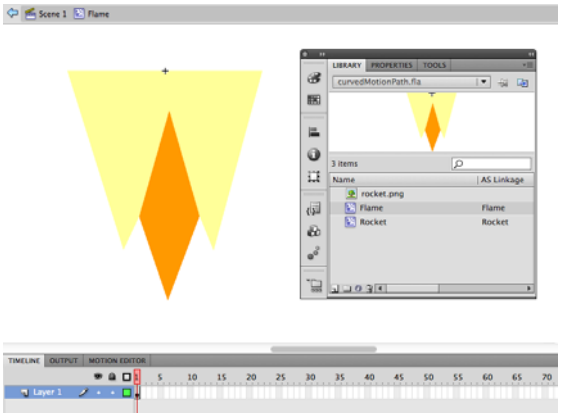

Figure 11-24. Frame one of the flame symbol.

Next, we're going to duplicate this image on frame 5 and make some changes to it.

11. Select frame 5 in the Timeline. Right-click on it and choose **`Insert keyframe`**. A new black keyframe will be added to frame 5, and you'll see that your drawing will also have been copied over to that frame. It's also completely selected, which you can tell by the pattern of tiny dots over it. You now have two copies of the same drawing: the first on frame 1, and the second on frame 5. Figure 11-25 shows what your Timeline should now look like.

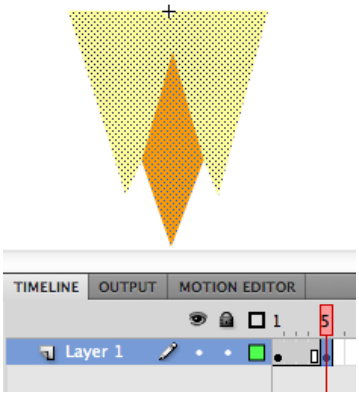

Figure 11-25. When you insert a new keyframe, all the graphics from the previous keyframe are automatically copied into it.

12. Make some changes to the colors of the flame on this frame. I've reversed the yellow and orange areas of color in my drawing.

13. Select frame 9 in the timeline, and insert a new keyframe. (Right-click and choose **`Insert keyframe`** from the option list.) A black keyframe will appear on frame 9, and the drawing from frame 5 will be copied over, as you can see in Figure 11-27.

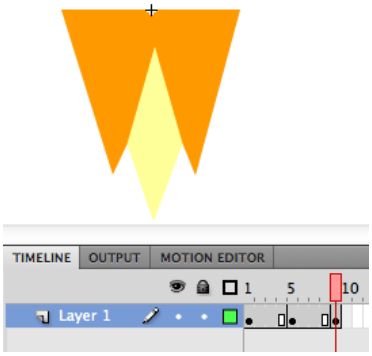

Figure 11-27. Insert a new keyframe on frame 9.

We now have a flickering flame effect. You can test this by clicking the **`Loop`** button at the bottom of the Timeline and then clicking the **`Play`** button, shown in Figure 11-28. The animation will loop continuously between frames 1 and 9.

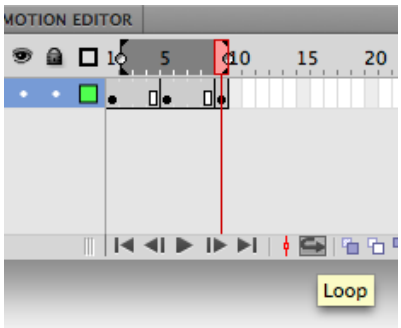

Figure 11-28. Click the Loop button and press play to test a looped animation.

Let's now add this flickering flame symbol to the rocket.

> *If you want to animate a drawing you've made along a motion path, you first have to convert the drawing to a symbol. To do this, select the entire drawing, right-click on it, and choose convert to symbol. You can convert it to either a Graphic, Movie Clip or Button symbol. Movie Clip symbols give you the most flexibility for controlling the animation with code later, but Graphic symbols are fine for simple motion path animations that you won't need to control with code. We'll be looking at how to create and use button symbols later in this chapter.*

Making composite objects

You can add instances of symbols inside other symbols to create composite objects that you can target with code. In the next few steps we'll add an instance of the Flame symbol to the rocket, and then write a program that switches the flame on and off at the click of the mouse.

1. Double-click on the **`Rocket`** symbol in the **`Library`** to enter the Rocket's symbol editing mode.
2. Create a new layer and give it the name **`flame`**.
3. Make sure that the first frame of the new layer is selected, and drag an instance of the **`Flame`** symbol onto the Rocket's stage.
4. With the flame still selected, open the Properties panel and give it the instance name **`flame`**. If you don't give it an instance name, you won't be able to access it with AS3.0 code. Figure 11-29 shows what the Rocket symbol's stage and Timeline should now look like.

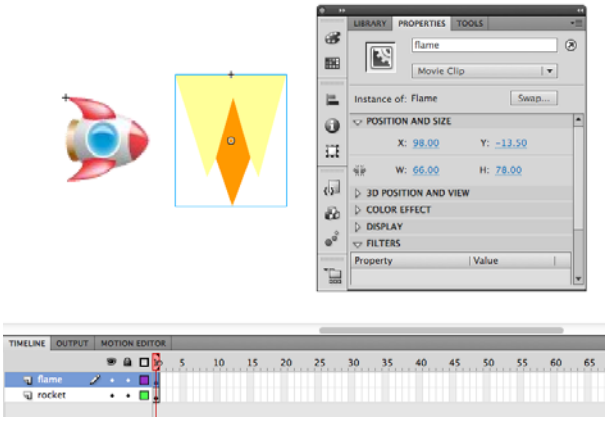

Figure 11-29. Add an instance of the Flame to the Rocket symbol and give it an instance name.

5. Use the **`Free Transform Tool`** to scale and rotate the flame so that it's the same size as and aligned with the rocket's engine. Then reposition the **`flame`** layer so that it's below the rocket layer, as shown in Figure 11-30.

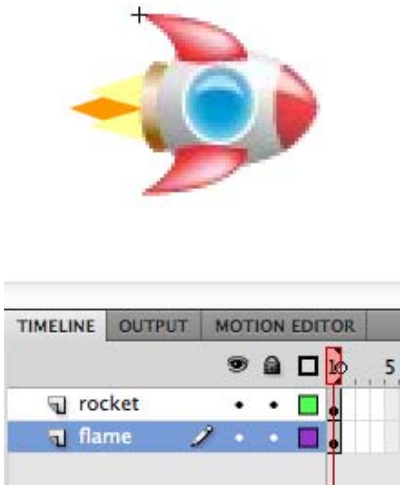

Figure 11-30. Resize and rotate the flame, then reposition its layer below the rocket.

6. Save the FLA file with the name `compositeObjects`.
7. publish the SWF file.

Let's next find out how you can target the `flame` sub-object with code.

Programming composite objects

Move over to Flash Builder, or whichever IDE you're using, and we'll write a program that lets us make the rocket's flame visible or invisible when we click the stage.

1. Create a new ActionScript project called `ProgrammingCompositeObjects`.
2. Create a folder called `swfs` in the project directory and copy the `compositeObjects.swf` file you published in the previous section into this folder.
3. Add the following ActionScript code. You'll find the complete working example in the `ProgrammingCompositeObjects` folder in the chapter's source files.

```
package
{
import flash.display.Loader;
import flash.display.MovieClip;
import flash.display.Sprite;
import flash.events.Event;
import flash.events.MouseEvent;
import flash.net.URLRequest;

  [SWF(width="550", height="400",
backgroundColor="#FFFFFF", frameRate="60")]

public class ProgrammingCompositeObjects extends Sprite
{
//Create the objects needed to load and display the SWF
private var _swfURL:URLRequest
= newURLRequest("../swfs/compositeObjects.swf");
private var _swfLoader:Loader = new Loader();
private var _animation:MovieClip;
```

```
public function ProgrammingCompositeObjects()
    {
//Load the SWF and add a listener that runs when
      //the file is loaded
     _swfLoader.load(_swfURL);
_swfLoader.contentLoaderInfo.addEventListener
(Event.COMPLETE, swfLoadHandler);

//A mouse event listener starts the animation
      //when the stage is clicked
stage.addEventListener
(MouseEvent.MOUSE_DOWN, mouseDownHandler);
    }
private function swfLoadHandler(event:Event):void
    {
      //Copy the event handler's currentTarget.content property
      //into the _animation MovieClip object
     _animation = event.currentTarget.content;

//Add the _animation to the stage
stage.addChild(_animation);
    }
private function mouseDownHandler(event:MouseEvent):void
    {
//Toggle the flame's visibility
_animation.rocket.flame.visible
= !_animation.rocket.flame.visible;
    }
  }
}
```

When you compile the program you'll see that rocket follows the motion path, and the flickering flame animation stays completely aligned with it. Clicking the stage toggles the flame's visibility on an off, as illustrated in Figure 11-31.

Figure 11-31. Click the stage to toggle the flame's visibility.

You can access the flame sub-object to change any of its properties with following ActionScript code:
`_animation.rocket.flame`

You can see what a powerful tool Flash Professional can be to quickly create complex game objects and animations. A simple effect like this looping rocket with its flickering flame took just minutes to make. Can you imagine what a complex and time consuming job it would be if you were building it entirely out of code? And of course, you can apply collision detection and all the other tricks you've learnt over the course of this book to any of these objects.

Learning more about animation

Game animation is very big area of specialization, and we've only just scratched the surface in this chapter. As simple as these example have been, they've covered all of the most important techniques and concepts and may well be all you need to know for most of your game projects. But there's much more you can learn, and the best place to start is Adobe's Animation Learning Guide, which you'll find here:

http://www.adobe.com/devnet/flash/learning_guide/animation.html

It goes into detail about more specialized techniques such as shape tweening, converting motion paths to AS3.0 code, and various customization and preset options.

Flash Professional also a special tool called the **`Bone Tool`** that you can use for making sophisticated character animation. Adobe has a detailed article about how to use it here:

http://www.adobe.com/devnet/flash/articles/character_animation_ik.html

Happy animating!

Making Buttons

Another nice feature of Flash Professional is that it lets you create a special Button symbol. Button symbols contain the three states, Up, Over and Down, that we discussed in chapter 4, so you don't have to make and program them with code. It can be time consuming to build buttons entirely out of AS3.0 code, so you'll save yourself a lot of work if you make the buttons for your game's user interface using Flash Professional.

The steps below show you how to make a button symbol with Flash Professional. We'll look at a practical example of how you can use it to make a play button for a game.

1. Create a new Flash Professional document called **`button`**.
2. Select **`Insert > New Symbol`** from Flash Professional's main menu.
3. Give it the name **`PlayButton`** and change its **`Type`** to **`Button`**.
4. Check the **`Export for ActionScript`** option and click the **`OK`** button to create the new symbol.

Your new **`PlayButton`** symbol will now be visible in the **`Library`**. Double click on it to enter symbol editing mode. The button's stage will be empty, but you'll see that it's got a special timeline, shown in Figure 11-32.

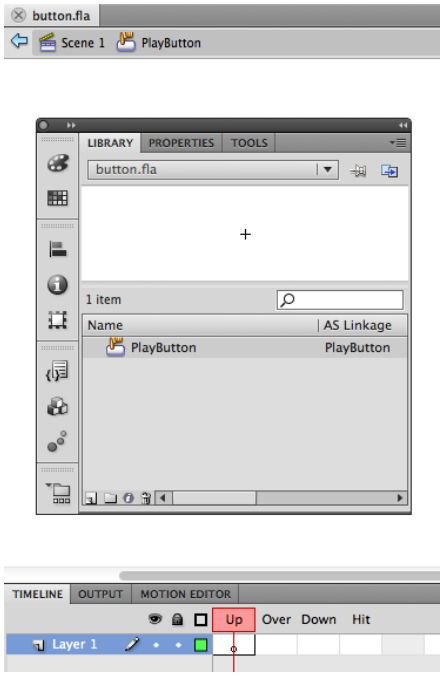

Figure 11-32. The Button symbol's timeline has 3 special frames that determine the button's states.

The first three frames are labeled **`Up`**, **`Over`** and **`Down`**. Whatever graphics you add to those frames will determine the up, over and down states of your button.

The fourth state, **`Hit`**, determines the area that the button is sensitive to. If you have a very small button, or an irregularly shaped one, you could draw a large rectangular shape in the **`Hit`** frame that would define the area and shape that the button is sensitive to. Any shape you draw in the Hit frame won't be visible, Flash just uses its size and shape to determine the button sensitivity area. If your buttons are large and square-ish, you probably won't need to use the Hit frame.

Let's draw the three button states on these frames:

5. Create a new layer, and name it **`text`**. Name the original bottom layer **`background`**.
6. Use Flash Professional's drawing tools to draw a rectangle on the **`Up`** frame of the **`background`** layer. Make sure you draw this in the center of the symbol, which is indicated by a black crosshair. (You can draw a rectangle with rounded corners using the **`Rectangle Primitive Tool`**.)
7. Use the **`Type Tool`** to add the word **`Play`** to the **`Up`** frame of the **`text`** layer. Your button should now look something like Figure 11-33.

Figure 11-33. Create the button's up state on two layers.

> *While working with Flash Professional's drawing tools, open the context sensitive Properties panel to change any of the tools properties. You'll find the options are similar to the kinds of tool options you're familiar with in Photoshop and Illustrator.*

8. To create the Over state, first insert keyframes on both the **`text`** and **`background`** layers of the **`Over`** frame. When you do this, the graphics from the **`Up`** frame will automatically be copied into the **`Over`** frame, as shown in Figure 11-34.

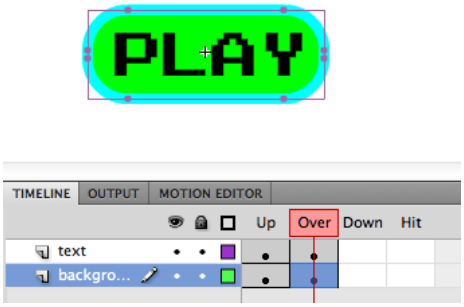

Figure 11-34. Insert keyframes in the Over frame.

9. Create a simple highlighting effect, which is easy to do just by making the rectangle's background a lighter color. I've change mine from green to bright yellow.

10. Add keyframes on both layers of the **`Down`** frame. The graphics from the **`Over`** frame will be copied over. Change the graphics in some way that make it look like it's being pressed. Changing the rectangle's background color to a dark blue is a quick way to do this. My **`PlayButton`** symbol now looks like Figure 11-35.

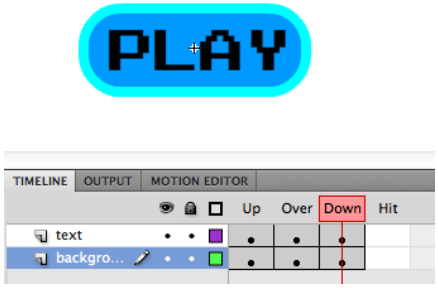

Figure 11-35. Create the Down state graphic.

My button now has three states. Let's combine it with a game title graphic next.

Creating a game tile SWF

In this next section we'll create a simple title graphic, and then add the title and button together on the main stage to create our game title SWF.

1. Create a new Movie Clip symbol called **`Title`**.

Draw a simple title graphic for your game, as you can see in Figure 11-36. You can also design your graphic in Photoshop and Illustrator and import it into the symbol for more control over the design.

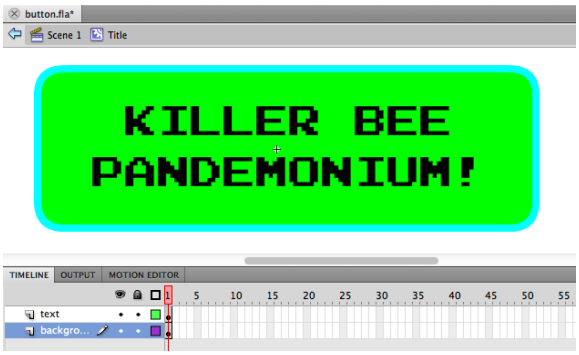

Figure 11-36. Create a game title.

2. Click the **`Scene 1`** button at the top left corner of the workspace to return to the main stage.

The game that I'm going to add this to has a stage width of 800 by 600 pixels. Let's change the dimensions of the Flash Professional document so that our published SWF will match this size.

3. Click anywhere on the empty main stage, then open the **`Properties`** panel. The stage properties are now accessible. Change the **`Size`** to 800 by 600 pixels, as shown in Figure 11-37. You can change all the document properties and also change the SWF publish settings from this panel.

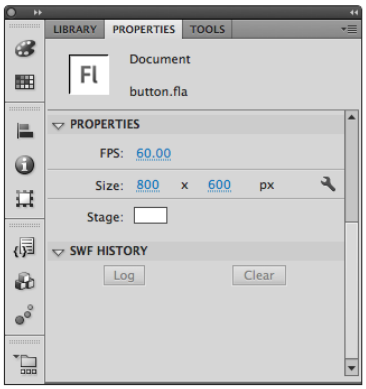

Figure 11-37. Change the stage's width and height in the stage properties panel.

Drag an instance of the **`PlayButton`** and **`Title`** symbols onto the main stage. Give the button the instance name **`playButton`**, as shown in Figure 11-38.

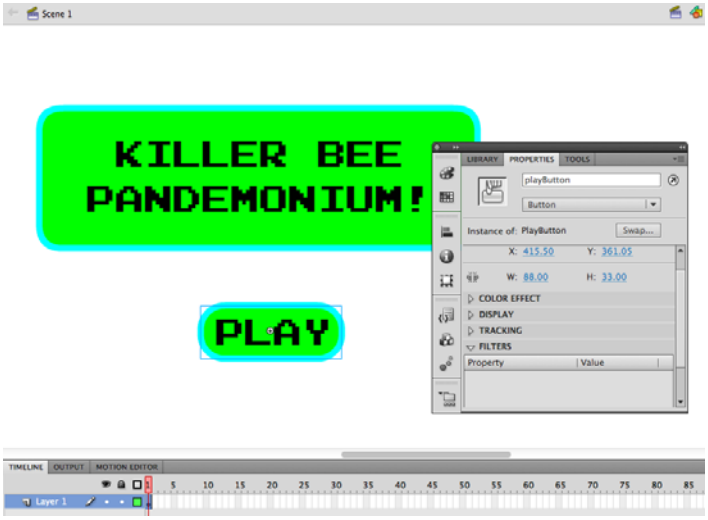

Figure 11-38. Drag instances of the symbols onto the main stage.

4. Save the Flash Professional document.

To see if your button is working and everything looks as it should, select **`Control > Test Movie > Test`** (Or use the well-worn shortcuts: **`Ctrl-Enter (Windows) or Command-Enter (Mac OSX)`**). The Flash Player will launch and show you what the SWF will look like. Click on the button and test that its three states are working as you expect them to.

5. Select **`File > Publish Settings`** and change the name of the published SWF to **`gameTitle.swf`**.

Click the **`Publish`** button in the **`Publish Settings`** window to publish the finished SWF file.

We can now load this SWF file into an AS3.0 and program the **`playButton`** instance so that it launches the game. Let's see an example of how that works next

Programming the play button

You'll find a project folder in the chapter's source files called **`ClickToPlay`**. It's the game Killer Bee Pandemonium! with the addition of the title page and start button that we created in the previous section. When the game loads, you'll first see the title and play button. Click the play button, and the game starts. Figure 11-39 illustrates what you'll see.

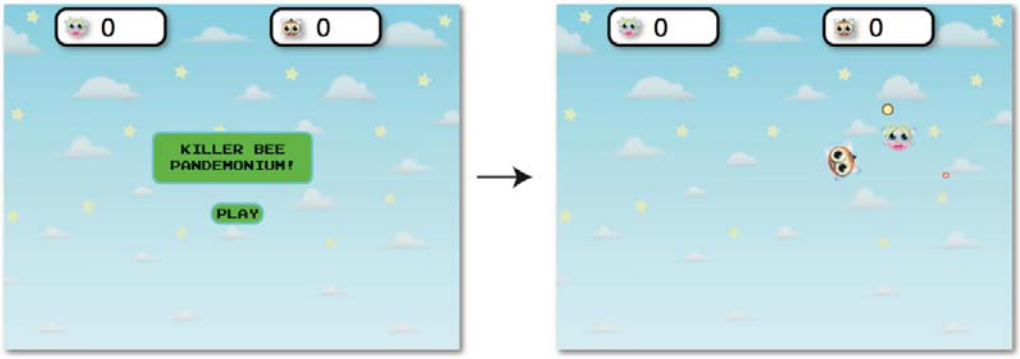

Figure 11-39. Click the button to play the game.

There's nothing new to learn about how any of this code works. It's just taking some techniques you already know and putting them together in a new way.

The first thing the application class does is to load the SWF file. We don't want the game to start before it's loaded.

```
public function ClickToPlay()
{
        _swfLoader.load(_swfURL);
        _swfLoader.contentLoaderInfo.addEventListener
          (Event.COMPLETE, swfLoadHandler);
}
```

When the SWF is loaded, the `COMPLETE` listener calls the `swfLoadHandler`. The job of the `swfLoadHandler` is to initialize all the game objects and add them to the stage. Most importantly, the **`swfLoadHandler`** adds the loaded SWF to the stage and adds a **`MOUSE_DOWN`** listener to the `playGame` button.

```
private function swfLoadHandler(event:Event):void
{
        //...game object initial settings...

        //Copy the event handler's currentTarget.content property
        //into the _titleMovieClip object
        _title = event.currentTarget.content;

        //Add the _title to the stage
        stage.addChild(_title);

        //Add a MOUSE_DOWN listener to the loaded SWF's button
        _title.playButton.addEventListener
    (MouseEvent.MOUSE_DOWN, playButtonHandler);
}
```

When the button is clicked, the `playButtonHandler` is called. Its job is to make the title SWF invisible, remove the `mouseDownHandler`, make the bee and fairy visible, and, most importantly, add the `ENTER_FRAME` listener to the stage so that the game can start playing.

```
private function playButtonHandler(event:MouseEvent):void
{
        _title.visible = false;
        _bee.visible = true;
        _fairy.visible = true;
        stage.addEventListener
          (Event.ENTER_FRAME, enterFrameHandler);
        _title.playButton.addEventListener
          (MouseEvent.MOUSE_DOWN, playButtonHandler)
}
```

All of this is routine code. You can see from this that as long as you know how to access the button inside the loaded SWF, you can program it exactly the same way you've programmed any other buttons.

If you're creating a game with a complex user interface, you can save yourself hours of work by designing it and setting it up in Flash Professional. Flash Professional has limited graphic design capabilities, but you can design all your button states and graphics in Photoshop or Illustrator and

import them into Flash Professional. Use Flash Professional to build symbols from these graphics, compose your UI on the main stage, and give all the buttons instance names. Then all you need to do is program those buttons in your AS3.0 program. When you come to programming the game, you won't have to deal with any of the substantial amount of code needed to setup, layout and program all those buttons and other interface elements. This chapter has shown you very simple examples of how all these pieces fit together, but it's exactly the same workflow you'd follow for a much bigger, more complex user interface.

Using Embedded SWF symbols

There's another great use for Flash Professional for making games. It's possible to create game objects in Flash Professional as symbols and then embed them, piece by piece, into your game. You can make a class from each symbol object, and access all of that object's properties in your game. This is a very powerful and flexible way to use Flash Professional as a library or storehouse for game objects. Let's take a quick look at how this works.

In the chapter's source files you'll find a FLA file called **`resourceLibrary.fla`**. It contains the same Rocket and Flame symbols that we've used in other examples. All the symbols have been set to **`export for ActionScript`**, just as they have been in other examples. The Rocket includes a sub-object of the Flame symbol with the instance name **`flame`**. None of the objects have been added to the main stage, they just exist in the Library. Figure 11-40 shows what the FLA file's Library looks like.

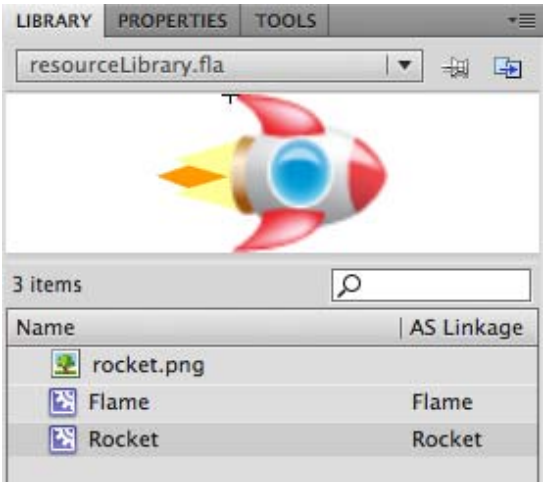

Figure 11-40. You can embed symbols from Flash Professional's Library directly into your program.

This FLA file was published as an SWF with the name resourceLibrary.swf. In the chapter's source files you'll find a project folder called **`EmbeddingAssets`**. Run the SWF and you'll see the rocket and the flame on the main stage, as shown in Figure 11-41. This might be spectacularly underwhelming, until you consider that these objects were made by extracting them individually from the Library of the embedded SWF file.

Figure 11-41. The objects have been extracted from the embedded SWF file's library.

The flame animation flickers, just as you would expect it to. Here's the `EmbeddingAssets` application class that embed the SWF, extracts the symbols from its Library, and then displays them on the stage.

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

  [SWF(width="550", height="400",
backgroundColor="#FFFFFF", frameRate="60")]

public class EmbeddingAssets extends Sprite
  {
//Embed the symbols from the SWF file
    [Embed(source="../swfs/resourceLibrary.swf", symbol="Rocket")]
private var Rocket:Class;

    [Embed(source="../swfs/resourceLibrary.swf", symbol="Flame")]
private var Flame:Class;

public function EmbeddingAssets()
    {
//Create a new rocket object
var rocket:Sprite = new Rocket();
addChild(rocket);
rocket.x = 200;
rocket.y = 175;

//Create a new flame object
var flame:MovieClip = new Flame();
addChild(flame);
flame.x = 300;
flame.y = 175;

      //flame.stop();

//Get access to rocket's flame sub object
var rocketFlame:MovieClip
= rocket.getChildByName("flame") as MovieClip;

      //rocketFlame.stop();
    }
  }
}
```

Let's first take a look at how the rocket is embedded and displayed.

```
[Embed(source="../swfs/resourceLibrary.swf", symbol="Rocket")]
private var Rocket:Class;
```

The `source` is the name of the SWF file that contains the Library symbols we want to access. symbol is the actual name of the symbol in the Library that you want to use in the game. In this case, it's the `Rocket` symbol.

It's then just a really simple matter of making an object from the `Rocket` symbol, like this:

```
var rocket:Sprite = new Rocket();
```

One quirk to this system is that the rocket has to be created as a Sprite. The reason for this is that, even through the Rocket symbol was created as a Movie Clip object, it only has one frame on the timeline. AS3.0's compiler will generate this error if you try and create it as a `MovieClip` object:

```
Error #1034: Type Coercion failed
```

The Rocket symbol has a sub-object called `flame`, which is the flickering flame animation. A reference to the flame object is embedded along with the Rocket symbol, but only as string. That means we have to use a special method called `getChildByName` to access the string and convert it into an object we can control with AS3.0 code. It also has to be cast as a `MovieClip` object so that we can control its timeline with `stop` and `play` methods. Here's the code that does all this:

```
var rocketFlame:MovieClip
  = rocket.getChildByName("flame") as MovieClip;
```

We can now control the rocket's flame sub-object with code, like this:

```
rocketFlame.stop();
```

This will stop the flame animation. You can use any of the other MovieClip methods and properties that we've looked at in this chapter to control the animation.

The Flame symbol is embedded in the same way, but because it has more than one frame on its timeline, we can create it as a MovieClip object, like this:

```
var flame:MovieClip = new Flame();
```

You can control its timeline, just as you would expect to, like this:

```
flame.stop();
```

This is a great system, because it means even if you don't want to use Flash Professional to make complex game interfaces, you can use it as a storehouse for game objects and animations. This is especially useful if you have lots of complex objects, with many sub-objects. Its quick, easy and fun to make those kinds of objects with Flash Professional, and you can see how easy it is to embed them and target them with code once they're made. There's also nothing stopping your from creating your game screens as symbols in Flash Professional and embedding and using them for your game's user interface in this same way.

Using SWC files

Instead of embedding or loading objects from SWF file, you can use Flash Professional to create something called an SWC file. An SWC file is a packaged collection of symbols and classes. You can

import SWC files into your AS3.0 programs and easily access all the symbols and classes they contain.

Here's how to export an SWC file from Flash Professional. It will create a package of all the symbols in the Library. (Make sure that your symbols are set to export for ActionScript.)

1. Right-click any symbol in Flash Professional's Library. Select **`Export SWC file`**.
2. Give the SWC file a name, such as GameObjects.swc. Choose a save location in your project directory and click **`OK`**.

Once you've created the SWC file you need to tell your IDE (Flash Builder or Flash Develop) where to look for it so that it can access it when it compiles your program. Here are the steps to follow in Flash Builder to do this:

3. Select Project > Properties.
4. Choose the **`ActionScript Build Path`** option on the left.
5. Click the **`Add SWC`** button.
6. Browse to the path the folder that contains the SWC file and click the **`Open`** button. The SWC file will now be added to the Build path libraries list.

You can now create an object from the SWC symbol directly like this:

```
var rocket:Rocket = new Rocket();
var flame:Flame = new Flame();
```

You don't have to type it as a Sprite or MovieClip. You also don't need to use `getChildByName` to access a sub-object. You can do it directly, like this:

```
rocket.flame.stop();
```

All these advantages mean that SWC files are the preferred way of using Flash Professional to create a game object Library. Also, because you can import almost any type of file into Flash Professional (sounds, videos and images) you can store and access all of your game's assets in this same way. Just create them as symbols in Flash Professional and export them for ActionScript in the symbol `Properties` panel. You won't have to do any loading or embedding. Your code will be much easier to look at and manage. The trade-off is that all your objects are trapped inside this SWC file. If you want to change any them, you'll need to make changes in Flash Professional and then re-export the SWC file.

Most libraries of classes that you might want to use for specialized game features, like 3D or physics, are also packaged as SWC files. You now know how to install and use them in your games.

> *SWC files are really just ZIP files with a different file extension. If you rename the ".swc" file extension to ".zip" you can decompress and view the contents of any SWC file.*

Learning more about Flash Professional

You can see that Flash Professional is a wonderful tool that can greatly speed up the development of your games. It works seamlessly with all the other technologies and software we've covered in this book, and is a lot of fun to use.

Of course, there's much, much more it can do, and it's even got a complete programming environment that you can use to do all of your AS3.0 game programming if you want to. The best place to start your further exploration of Flash Professional is at Adobe's official website here:

http://www.adobe.com/devnet/flash.html

You'll also find a range of books covering all aspects of working with Flash Professional at www.apress.com.

Publishing your games

If you want your game to be played over the internet, on desktop computers, and any mobile phone or tablet without having to write it more than once, there's currently only one way to do it: AS3.0 and Flash. Because AS3.0 is an interpreted programming language, your games and programs can run on any device that has an AS3.0 interpreter. For the web, the interpreter is the Flash Player. For the desktop and mobile devices, it's the AIR runtime. Adobe aggressively targets new platforms as soon as they appear, so you're unlikely to find any computing device that won't be able to run your game in some way.

In the next few chapters I'm going to show you how you can publish your games for the following platforms:

The web, using an SWF and HTML file.

Desktop computers, using an AIR runtime file.

Mobile platforms: iOS and Android phones and tablets.

For desktop computers and mobile platforms, you'll need to use either Flash Builder or Flash Professional. I'll explain the steps you need to follow using Flash Builder in this chapter.

> *It's also possible to create desktop and mobile applications using Adobe's command-line compiler. It's called the Air Development Tool (adt) and you can find out how to use it here:*
>
> *http://help.adobe.com/en_US/air/build/index.html*
>
> *The ADT is a free Java application that is part of the Adobe's AIR SDK. You can use it without needing a copy of either Flash Builder or Flash Professional.*

You'll also need to invest a bit of time into learning about the details how to get your finished file onto the device you're publishing for. These details are constantly changing, and each platform has its own little quirks that you'll have to navigate around. But I won't push you into the wilderness without a packed lunch: this chapter will point to toward all the resources you'll need to help you solve the problems you're likely to encounter.

> *Because AS3.0 is an interpreted programming language, it means that the compiled program doesn't directly run on the CPU of the computer or device. It runs in the interpreter, which is software that translates the AS3.0 program into binary code (ones and zeros) that the CPU can understand. If the CPU has to invest more power into doing this translation, it has less power to play your game. That could result in less responsiveness or a stuttering and slowing down of the game.*
>
> *The solution to this is to do a lot of testing on the computer platform you're expecting to publish your game on. If you find certain aspects of the game are causing it to slow down or stutter, take a careful look at your code and graphics to find out what could be causing the problem. Make some changes and test it again. Adobe is also constantly improving its software*

interpreters, so make sure that you have an up-to-date version of the latest Flex SDK and Flash Player, and up-to-date packagers for the mobile devices you're publishing to.

Publishing for the web

Over 2 billion people worldwide have access to the web. If you want your game to be played by the largest number of people possible, then publishing for web is still the best way to reach them. It's also the quickest and easiest way to publish your game.

When you create an new ActionScript project in Flash builder, the project configuration wizard asks you what type of application you want to make: **`Web`** or **`Desktop`** as you can see in Figure 11-42.

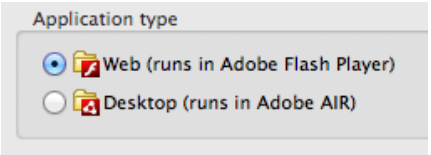

Figure 11-42. Choose the Web application option to publish your game to the web.

Choose `Web` and Flash Builder will automatically create all the files you need to upload your game onto the web.

When you're finished making your game, create a **release build**. Release builds make your finished file smaller by removing trace statements and debug information. You make a release build in Flash Builder by selecting **`Project > Export Release Build`** from the main menu. Flash Builder then automatically creates a folder called **`bin-release`** in your project folder. It contains all the files and support folders you need to upload your game to a web server. Figure 11-43 shows an example.

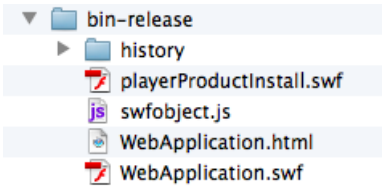

Figure 11-43. The bin-release folder contains all the files you need to upload your game onto the web.

The most important of these files are the SWF file, which is your finished game, and the HTML file, which is a web page that contains code that embeds the SWF file into it. HTML and CSS are the two technologies that are used together to make web pages. If you know how to program HTML and CSS, you can customize the web page that loads your game in any design or style you like.

To get these files onto the internet, you need the following:

- A web hosting company that will give you space on the web to upload your files. There are some free services, but the best charge a small monthly fee.
- FTP (File Transfer Protocol) software (Often called an FTP *client*). This is software that lets you upload files from your computer to the folder on the web. Your web hosting company will give you all the information you need to configure the FTP software so that it uploads your

files into the right directory. Most web hosting companies also have a web-based FTP application that you can use to do this as well.

- Optionally, you need a website address, like www.anywebsitename.com. Website addresses are called **domain names**. Your web hosting company will likely be able to help you register a domain name, but there are many domain name registration services on the web that you can use. Your web hosting company will tell you how you can to link your domain name with your root web folder. This lets the rest of the world find your game if they enter your website address.

Once you have your web hosting account set up, upload the files from the **`bin-release`** folder into your website's the root directory. Finally, change the name of the HTML file to **`index.html`**. Now whenever anyone types in your website address, the **`index.html`** page will automatically load the SWF file that contains your game. Any web browser that has the latest Flash Player plug-in installed will be able to play it.

Publishing desktop applications

You can distribute your Flash game as a stand-alone application that will run on any computer that has the AIR runtime installed. The AIR runtime is an interpreter that runs AIR applications. An AIR application will automatically try and download and install the air runtime when it's first launched, so users don't necessarily need to have it installed right away to play your game. Here's are the steps you need to follow create an AIR desktop application project.

1. In Flash Builder, select **`File > New ActionScript Project`**.
2. Select **`Desktop (Runs in Adobe AIR)`** as the **`Application Type`**.
3. Finish setting up and creating the project in the same way would for any other ActionScript project.

In the project's **`src`** folder you'll find a new XML file. The is the **application descriptor file**. It will have the name "ApplicationName-app.xml". It contains detailed information about the application, such as its window dimensions, what it needs to access on the operating system to run, and how the application should be displayed. You'll likely never need to change anything in this file, but in case you do, you can read all about it in the chapter "AIR application descriptor files" at Adobe's AIR website here: http://help.adobe.com/en_US/air/build/index.html

4. Program, test and debug your game exactly as you would for any other ActionScript project.
5. To create your finished AIR application, select **`Project > Export Release Build`**. However, before it builds the finished application, Flash Builder will ask you to digitally sign it. An AIR application has to be signed before it can be installed on another computer besides your own. You can either create your own digital signature, or have it signed through a third-party digital signature company like VeriSign, Thawte, GlobalSign, or ChosenSecurity.

Here's how to create your own digital signature to export the finished AIR application:

1. Select either the **`Signed AIR package`** or **`Signed native installer`** option from the Export Release Build window. The **`Signed AIR package`** will create a generic AIR application that can run on either Windows or OSX. The **`Signed native installer`** option will creates a platform specific installer: either Windows or OSX. This will depend on the operating system you're running Flash Builder on.
2. Click the **`Next`** button. The **`Package Settings`** window appears.
3. Click the **`Create`** button, to create a new digital certificate. (If you've already created one previously, choose Browse to select it.)

A new window will open that asks you to enter your name, organization, country and a password for the certificate. You also need to choose between 1024 and 2048 bit encryption – either will do. Fill this information out, give the certificate a name, and save it by clicking the **OK** button. (If you need to change this certificate at a later time, you can return to this page by selecting `Project > Properties > ActionScript Build` **Packages**)

4. Click Finish in the Export Release Build window.

An AIR application installer package will be created in your project directory.

> *To have the application signed by a third party digital certificate company, you need to select* ***Export an Intermediate AIRI File that will be Signed Later****. However, these can only be signed using the Air Development Tool, which is a specialized command-line based software. If you need to do this, you can find out more about it here: http://livedocs.adobe.com/flex/3/html/help.html?content=CommandLineTools_5.html.However, it's very unlikely you'll ever need to have your file signed by a third party certificate company, unless you're working for an organization that absolutely requires this.*

After you've exported a release build of your AIR application, you'll find the installer package in your project directory. It will have a ".air" extension if you select the **Signed AIR package** option. It will have a ".dmg" extension (OSX) or ".exe" (Windows) extension if you chose the **Signed native installer** option. Double-click any these files to install and run the game. The installer will automatically download and install the latest AIR runtime to make sure your game has access to all the software components it needs to run properly.

If you're using Flash Builder 4.6 or later, it has an additional export option called **Signed application with AIR runtime bundled**. This option bundles the application together with the AIR runtime in a single file. It means that application can run without having to be connected to the internet to download the latest runtime.

If you want to make a custom icon for your AIR application, design a square icon graphic in Photoshop or Illustrator. You'll need to export it as a PNG file in 4 different sizes: 16x16, 32x32, 48x48, and 128x128. Save these PNG files in a folder called **icons** in your project directory. Then you need to update the application descriptor file so that it knows where to find these icon images. Look for the XML tag called **<icon>**. Enter the file path to each of the images, for each size you've created, like this:

```
<icon>
<image16x16>../icons/icon16.png</image16x16>
<image32x32>../icons/icon32.png</image32x32>
<image48x48>../icons/icon48.png</image48x48>
<image128x128>../icons/icon128.png</image128x128>
</icon>
```

Your AIR application will select the correct sized icon for your target platform when you create the release build of your application.

For more detailed information about AIR applications, see Adobe's AIR website here: http://help.adobe.com/en_US/air/build/index.html

Publishing for mobile devices

It's as easy to build test and publish games for mobile devices using Flash Builder as it is for any other platform. Flash Builder simulates the device you're building for, so you don't even need access to the actual device while you're programming the game. Adobe is constantly updating Flash Builder's device

profile list as new mobile devices become available, so you're likely to find a profile for any device you want to make a game for.

> *You can also use Adobe's Device Central software to test your game in detailed simulations of different devices. You can find out more about Device Central here: http://www.adobe.com/products/devicecentral.html*

In theory it's possible to write one game program for all devices. But in practice you're going to encounter problems working with different screen resolutions and device capabilities. The best strategy is to target a single device, get your game working properly on it, and then gradually work in code that selectively tests for the capabilities of other devices. This means a lot of testing on the devices that you're writing for, and you'll need to invest a bit of time to find out what those devices can do. Does the device have an accelerometer, a GPS or a camera? You can use some AS3.0 code that will tell you this, and you can then selectively run code if a certain capability is supported. You can find out all about this in the chapter "ActionScript 3.0 API support for mobile devices" in Adobe's document "Building Adobe AIR Application with the Packager for iPhone":

http://help.adobe.com/en_US/as3/iphone/index.html

Of course, the phones and tablets that your games are going to be played on use a touch interface. Moving your finger across the screen will have the same effect as moving the mouse, and a finger tap will be interpreted as a mouse click. But your mobile game is created using the AIR runtime, and that means it also has access to AS3.0 libraries that support **multitouch** and **gestures**. Multitouch is touching the screen with more than one finger, and a gesture is a specific sequence of touch-based movements, such as using two fingers on the screen to rotate an object. You can find all the code you need to implement multitouch and gestures in the chapter "Touch, multitouch and gesture input" from the "User Interaction" chapter of Adobe's ActionScript 3.0 developer's guide:

http://help.adobe.com/en_US/as3/dev/index.html

You can find out many more details on the specifics of publishing mobile games in Adobe's online document, "Developing Mobile Applications with Flex and Flash Builder":

http://help.adobe.com/en_US/flex/mobileapps/index.html

Adobe also has a dedicated website to developing mobile apps and games where you'll find links to the latest articles, tutorials and procedures related to mobile development with Flash and AS3.0:

http://www.adobe.com/devnet/devices.html

But you'll be happy to know that if you've made it this far in this book, your AS3.0 skills will be perfectly adequate to implement all these new features. And the actual process of setting up and debugging a mobile project with Flash Builder couldn't be easier.

Creating an ActionScript mobile project

Follow these steps to create an ActionScript mobile project with Flash Builder. You're actually creating a specialized AIR application, so the steps are very similar to those you need to follow when creating a desktop app.

1. Select File > New > ActionScript Mobile Project.
2. Give your project a name and choose a location for the project files. Click the **`Next`** button.

The Mobile Settings page opens and asks you to select your target platforms. In Flash Builder 4.5 the two options are Apple iOS and Android. You can select both of these if you're building a multi-platform game. But I recommend just developing for one platform at a time until you fully understand what both

are capable of and you understand the more complex code you'll need to write to build a multi-platform game.

The **`Application settings`** options at the bottom of this page lets you choose between two screen orientation options: **`Automatically reorient`** and **`Full screen`**. These options affect settings in the application descriptor file, and you can always change them in that file later. You can also use AS3 code directly in your game to set the screen orientation.

Figure 11-44 shows what the Mobile Settings page looks like.

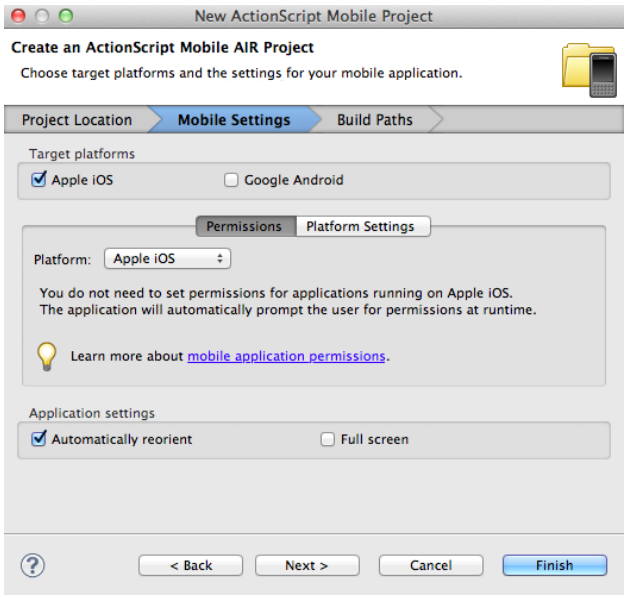

Figure 11-44. Select a target platform for your mobile project.

3. Click the **`Platform Settings`** button to choose a specific device. If your target platform is Apple iOS, you can also choose between the iPhone/iPod Touch and iPad in the **`Target devices`** option menu. Figure 11-45 illustrates this.

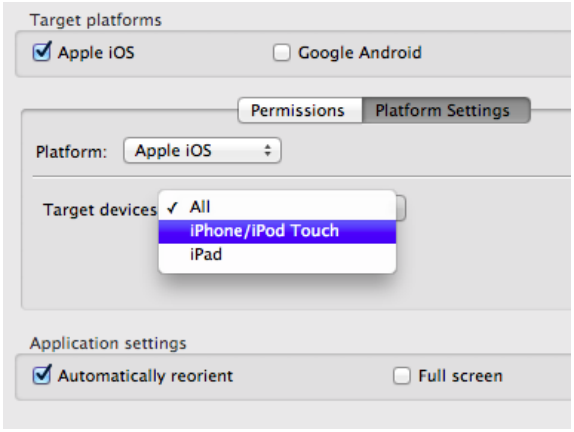

Figure 11-45. Target a specific device.

4. Click the **`Next`** button. You'll see the **`Build Path`** page, which lets you link the project to any class libraries you may have. Click the **`Finish`** button and the project will be created.

You can now start programming your game, just as you would for any other ActionScript project. Flash Builder generates a bit of code to get you started, which looks like this:

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

public class MobileApplication extends Sprite
{
public function MobileApplication()
{
super();

// supportautoOrients
stage.align = StageAlign.TOP_LEFT;
stage.scaleMode = StageScaleMode.NO_SCALE;
}
}
}
```

You can ignore (and delete) the `super()`method. The two additional stage property options are what you need to include if you want your game to automatically change screen orientation when the user of the device rotates it. If you want to lock the orientation, which is a good idea for many types of games, you can use this directive:

```
stage.autoOrients = false;
```

You can find out much more about stage orientation and how to control it in the section "Stage orientation" from the "Display programming" chapter of Adobe's "ActionScript 3.0 Developer's Guide" (http://help.adobe.com/en_US/as3/dev/index.html).

Debugging a mobile project

Flash Builder simulates the device you're programming for, so you don't need access to the actual device or any other software to do your initial testing. When you compile the project for the first time, Flash Builder will open the **`Create manage and run configurations window`**. This lets you choose the target platform, and whether you want to test on the actual device or on the desktop. Figure 11-46 shows what this window looks like.

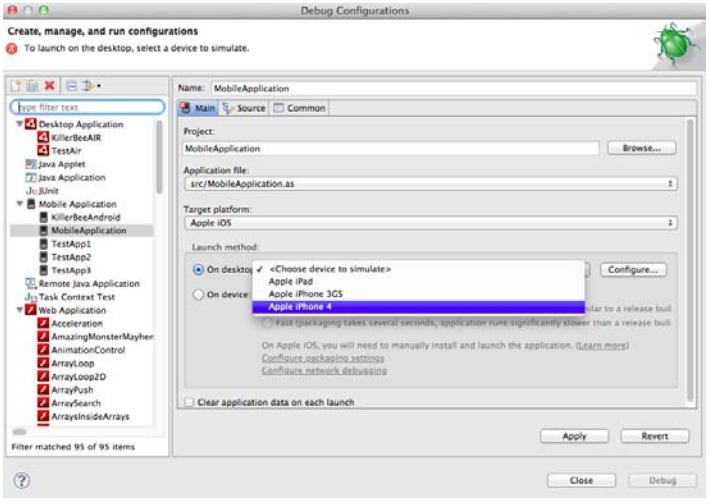

Figure 11-46. Configure the testing environment.

If you don't see the device that you're programming for, click the **`Configure`** button. This opens a list of devices that you can choose to test for, as shown in Figure 11-47. Adobe is constantly adding support for new devices as soon as they become available, so if you don't see the device you're testing for, click the **`Import`** button. Flash Builder will contact Adobe's website and download the latest device configurations.

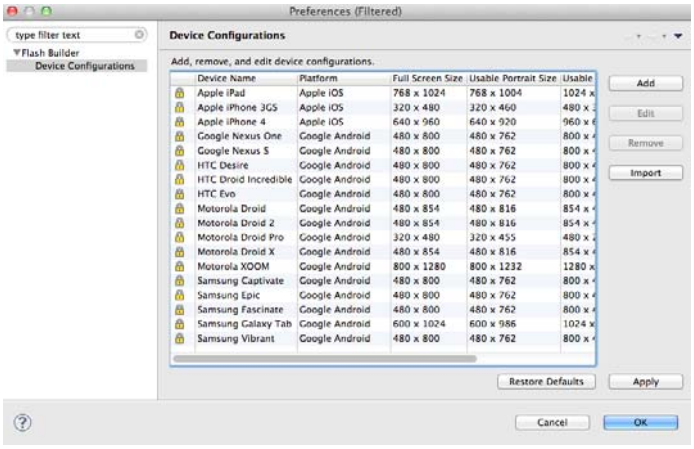

Figure 11-47. Select the device to test for.

This list also tells you screen sizes of each device, which you'll need to know for designing your game graphics.

> *You can make changes to these configurations at any time during the development of your game by selecting Run > Run Configurations from Flash Builder's main menu.*

Once you've setup the device configurations, click the **`Debug`** button, and Flash Builder will open a window that matches the size of the device you're testing for, and run your game in it. You can test and debug it as you would for any other AS3.0 project.

Deploying mobile games

The steps to deploying your game on mobile devices are different for each device and constantly changing. Make sure you check the latest documentation for the device you're writing your game for so you have the most up-to-date information. The chapter "Package and export a mobile application" from Adobe's online document "Developing Mobile Applications with Flex and Flash Builder" will contain the most recent procedures (http://help.adobe.com/en_US/flex/mobileapps/index.html).

In the following sections you'll find the general steps you'll need to follow to deploy games for iOS and Android devices at time of publishing.

Deploying your game on Android devices

Here's how to deploy your game on Android devices:

Select **`Project > Export Release Build`** from Flash Builder's main menu.

Choose the folder you want to export the final build to, and select **`Signed packages for each target platform`**. Click the **`Next`** button.

You'll be asked to provide a digital certificate, and you can either create a new one or use the same one you've used for any AIR desktop applications you might have created. On the **`Deployment`** page, you have the option of selecting **`Install and launch application on any connected devices`**. If you have an Android device connected to your computer via USB, Flash Builder will find it, copy the compiled application onto it, and launch it.

Click the **`Finish`** button and you'll find an APK file in your project directory.

To launch your game app on an Android device, copy the APK file onto the device. Run it using a file browser, and you'll then see your app in the device's **`Applications`** listing.

At the time of publishing, you have two good places to distribute and sell your game:

The Android Market:

https://market.android.com/publish.

Amazon's App store for Android:

http://www.amazon.com/mobile-apps/b?ie=UTF8&node=2350149011

A web search will bring up many more options for selling and distributing your Android game.

Deploying your game on iOS devices

The biggest hurdle to deploying an iOS app is the paperwork involved. You need to register as apple developer and pay a $99 for iOS developer and distribution certificates. You'll need to convert the iOS certificates to a P12 certificate so that Flash Builder can digitally sign your app. You also need a developer provisioning profile to deploy the app on a device, and a distribution provisioning profile to sell your game in Apple's app store. Yes, there are a lot of hoops to jump through!

You can register as an Apple developer at Apple's website here:

http://developer.apple.com/programs/register/

The steps involved in obtaining an iOS developer/distribution certificate and converting it to a P12 certificate are intricate and change frequently. Adobe has a detailed step-by-step listing of the current steps you need to follow that you can find in the chapter "Apple iOS development process using Flash Builder" in the document "Developing Mobile Applications with Flex and Flash Builder". These procedures could change at any time, so keep an eye on them.

Once you've got your P12 certificate, here are the steps you need to follow to run your application on an iOS device and submit it to Apple's app store.

1. Connect the iOS device to your computer and launch Apple's iTunes software.
2. Select **`Project > Export Release Build`** from Flash Builder's main menu. Select **`Signed packages for each platform`** and click the **`Next`** button.
3. Select the P12 certificate file that you obtained from Apple and enter the certificate's password.
4. Click the **`Finish`** button, and Flash Builder will compile the app and place it in your project folder. iOS apps have a ".ipa" extension.

In the next few steps you need to load your compiled IPA file into iTunes so that you can launch it on your device.

5. In iTunes, select **`File > Add to Library`**. Find the mobile provisioning certificate file that you would have obtained from Apple. It will have a ".mobileprovision" file extension.
6. Select **`File > Add to Library`** again and choose the IPA file that Flash Builder created.
7. Select **`File > Sync`** and your app will be appear on the connected device.

To submit your game to Apple's app store, you'll first need to re-build your app with Flash Builder. Supply Flash Builder with the distribution certificate and distribution provisioning profile that you obtained from Apple. Visit the iOS development center and log in to your account (http://developer.apple.com/devcenter/ios/index.action). Navigate to the iTunes Connect section. You'll be able to add your new app in the Manage Your Apps section. Make sure you also submit any support files, like your app icon PNG images, that Apple might require. Once you've submitted your app, you'll need to wait for approval from Apple before the your app appears in the app store.

These procedures are also highly subject to change, but Adobe has a guide to current app store submission procedures that you'll find here:

http://www.adobe.com/devnet/flash/articles/app_store_guide.html

Adobe also publishes a very helpful document specifically for iOS development with AS3.0 that you can find here: http://help.adobe.com/en_US/as3/iphone/index.html.

Summary

Although you've accomplished so much already, there's a great universe of learning ahead of you. Here's a quick roundup of some of the areas you might want to explore:

- **Adobe's online help documentation**: Adobe maintains excellent online documents detailing most aspects of AS3.0 programming. You can access these documents by pointing your web browser to Adobe's ActionScript Technology Center: http://www.adobe.com/devnet/actionscript.html. There you'll find links to the ActionScript 3 Developer's Guide and ActionScript 3 Reference for the Flash Platform. I've made numerous references to sections of these documents throughout this book, and they remain the most comprehensive primary source for all things Flash and AS3.0. Although many of the topics they deal with and the approaches they take are reasonably advanced, they should match your current level of ability. Spend some time going through some of the hundreds of articles and you're sure to find great food to fuel your developing programming skills.
- **3D**: With the release of Flash Player 11 and Stage 3D, Flash has become a superb platform for building interactive 3D worlds and games. You can find out how to program 3D games with AS3.0 at Adobe's Stage 3D website here: http://www.adobe.com/devnet/flashplayer/stage3d.html. There are some 3D game engines you can use to help you 3D games with AS3.0 and Flash: Alternativa, Flare3D, Away 3D and Adobe's own Proscenium. You can also use Stage 3D to make 2D games with full access to the computer or mobile devices Graphics Processing Unit (GPU) for great performance. The easy-to-use Starling framework can help you do this, as well as ND2D.
- **Physics**: There are some excellent packages of classes available for doing precise physics simulations with AS3.0. They include Box2D, Motor2, APE, Foam, Flave, World Construction Kit, JigLib and Glaze.
- **Tile-based games**: A style of game design in which all objects on the stage are assigned to squares in a grid. The code then creates an abstract model of the game behind the scenes in arrays, and that model is used to update the objects on the stage. Board games, strategy games, and certain puzzle games really benefit from the tile- based approach. Tile- based games

run very efficiently, use relatively little CPU power and memory, and are easy to extend if you want to add new game levels. Because of this, the tile- based approach also works well for platform and adventure games.

- **Vector math and geometry**: If you'll create games that involve some degree of physics simulation, it is really beneficial to spend a bit of time learning some vector math and 2D geometry. Even if you don't think you're good at math, give it a try—you might just surprise yourself. You can immediately apply what you learn to your games.
- **Saving game data**: If you want to save some data in your game (such as a score or a game level) so the player can quit the game and continue it later, you need to create a **shared object.** This is covered in detail in the subchapter "Storing local data" in the chapter "Networking and Communication" from Adobe's online document *Programming ActionScript 3.0 for Flash*.
- **Multiplayer games**: It's always more fun to play games with friends. To make multiplayer games you need to store game data on a server on the Web so that other players can access that data to update their own game screens. There are quite a few ways you can do this, but all require some research into additional "server- side technologies." Your Flash games will be able to communicate with these server- side technologies, but to implement most of them you'll need to learn new programming languages such as PHP or Java. Do a bit of research into Java socket servers such as SmartFoxServer, ElectroServer, and Pulse, among many others. (Java is very closely related to AS3.0, so you won't find it completely unfamiliar.) You can create a high- score list for your games using PHP, and you can store game data using a MySQL database. You can avoid Java and PHP directly by storing game data in a content management system (CMS) and accessing the data in Flash using XML. Adobe also provides its own Flash Media Servers. The Media Development Server is free, and although limited, is a great way to get your feet wet with multiplayer technologies in a familiar environment. As you can see, there's a lot to learn! But it's all well worth the time you'll put into it.

But of course, this book isn't really finished – you're only half way through. Advanced Game Design with Flash picks up where this book leaves off, and covers all the advanced techniques you'll need take your skills to a truly professional level.

As every game designer knows, making games is much more fun than playing them. Like an artist who's just learned how to mix paints and sketch out a few simple scenes, a bit of practice is all you'll need, and you'll be well on your way to creating that masterpiece. You've now got a solid foundation in game design with Flash and AS3.0—go and make some great games!