Gary Rosenzweig's

ActionScript 3.0 Game Programming University

Covers Adobe® Flash CS3 Professional

QUE

Gary Rosenzweig

GARY ROSENZWEIG'S

ActionScript 3.0 Game Programming University

Gary Rosenzweig

que®

800 East 96th Street
Indianapolis, Indiana 46240 USA

ActionScript 3.0 Game Programming University

ISBN-13: 978-0-7897-3702-1
ISBN-10: 0-7897-3702-7

Library of Congress Cataloging-in-Publication Data:
Rosenzweig, Gary.
ActionScript 3.0 game programming university / Gary Rosenzweig.
p. cm.
ISBN 0-7897-3702-7
1. Computer games—Programming. 2. Computer animation. 3. ActionScript (Computer program language) I. Title.
QA76.76.C672R73 2007
794.8'1526—dc22
2007028659

Printed in the United States of America
First Printing: August 2007

Associate Publisher
Greg Wiegand

Acquisitions Editor
Laura Norman

Development Editor
Laura Norman

Managing Editor
Gina Kanouse

Project Editor
Anne Goebel

Copy Editor
Keith Cline

Senior Indexer
Cheryl Lenser

Proofreader
Heather Waye Arle

Technical Editor
Cheryl Brumbaugh-Duncan

Publishing Coordinator
Cindy Teeters

Book Designer
Anne Jones

Senior Compositor
Gloria Schurick

Graphics
William Follett
Eve Park

Contents at a Glance

Contents

About the Author

As a youngster, **Gary Rosenzweig** was allowed to play video games whenever he wanted, as long as his homework was done first. His parents got him an Atari 2600 and an assortment of games. He loved to play Adventure, Asteroids, Pitfall, Raiders of the Lost Ark, and even that dreadful E.T. game.

At age 13, in 1983, his grandmother gave him a new TRS-80 Model III. The first thing he did with it was learn to program. And then, make games. He made some text adventure games, and then some RPG games, and then some arcade games. He was allowed to stay up all night making games, as long as his homework was done first.

In high school, Gary got to play with the Apple II computers pretty much whenever he wanted, as long as his schoolwork was done first. He made space shuttle simulators and spreadsheet programs. And some games.

Gary went on to study computer science in college at Drexel University. There he was told that with his degree, he could go on to be a programmer at any high-tech firm making business applications. But he wanted to make games, even if it was on the side, after he got his work done first.

After a side trip to get a Master's degree in journalism and mass communication from the University of North Carolina in Chapel Hill, Gary ended up getting a job where he could make games for kids using Macromedia Director.

Then, they invented the Internet. It was soon followed by Shockwave, a way to play Director content in web pages. Gary started making his own games for his own website in the evening, after his work was done first.

In 1996, Gary started his own company, CleverMedia, to produce games for the Web. He was soon creating both Shockwave and Flash games with some of the most creative people he ever met. CleverMedia and its sites grew over the years to become the single largest collection of web-based games by a single company. Gary has created more than 300 games in the past 12 years, most of which can be found at CleverMedia's main game site, www.GameScene.com.

Gary also likes to share what he knows. His sites http://FlashGameU.com, www.Director-Online.com, and www.DeveloperDispatch.com contain information for other developers. He has also written many books, including *Macromedia Flash MX ActionScript for Fun & Games, Special Edition Using Director MX,* and *Advanced Lingo for Games.* Gary wrote this book mostly on evenings and weekends, after his other work was done first.

Gary lives in Denver, Colorado, with his wife, Debby, and daughter, Luna. Debby and Gary also own The Attic Bookstore, an unusual used bookstore in Englewood, Colorado. Luna is only 5 years old, but is already playing games on her Macintosh computer, after her homework is done first, of course.

Dedication

A person should aspire to do three things with his career and life: do something he loves, do it better than anyone else, and make the world a better place. It is hard to find examples of someone who has accomplished all three. Steve Irwin was one such individual. He was an inspiration to me.

Acknowledgments

Thanks to everyone at CleverMedia: David Feldman, William Follett, Kevin Frutiger, Layle McFatridge, Eve Park, and Jay Shaffer. And to John Nyquist for his help with this book.

Thanks to the good people at Adobe and the Flash development team. ActionScript 3.0 rocks.

Thanks to my family: Debby Rosenzweig, Luna Rosenzweig, Jacqueline Rosenzweig, Jerry Rosenzweig, Larry Rosenzweig, Tara Rosenzweig, Rebecca Jacob, Barbara Shifrin, Richard Shifrin, Phyllis Shifrin, Barbara Shifrin, Tage Thomsen, Anne Thomsen, Andrea Thomsen, and Sami Balestri.

Thanks also to everyone at Que and Pearson Education for their hard work on this book.

We Want to Hear from You!

As the reader of this book, *you* are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

As an associate publisher for Que Publishing, I welcome your comments. You can email or write me directly to let me know what you did or didn't like about this book—as well as what we can do to make our books better.

Please note that I cannot help you with technical problems related to the topic of this book. We do have a User Services group, however, where I will forward specific technical questions related to the book.

When you write, please be sure to include this book's title and author as well as your name, email address, and phone number. I will carefully review your comments and share them with the author and editors who worked on the book.

Email:	feedback@quepublishing.com
Mail:	Greg Wiegand Associate Publisher Que Publishing 800 East 96th Street Indianapolis, IN 46240 USA

Reader Services

Visit our website and register this book at www.quepublishing.com/register for convenient access to any updates, downloads, or errata that might be available for this book.

Introduction

- Flash and Game Development
- Who Is This Book For?
- What Do You Need To Use This Book?
- Using the Example Games in Your Projects
- What You'll Find in This Book
- The FlashGameU.com Website

This is a great time to be a Flash game developer. Right now, there is no better development tool for small and medium-size games.

Flash CS3 Professional (a.k.a. Flash 9) is fast, powerful, and easy to develop with. The key to this is ActionScript 3.0, the great new programming language inside this latest version of Flash.

ActionScript 1.0 and 2.0 were often frustrating for game developers. They weren't fast enough to get key tasks done, and odd bugs and unexpected behavior often slowed down production.

ActionScript 3.0 is a very different animal. You'll find yourself developing quickly and effortlessly. Things just work, and work well. And the speed of ActionScript 3.0 will make your game design ideas work just as you imagined them.

Let this book become your guide to Flash game development. I hope you enjoy learning from this book as much as I enjoyed writing it.

Flash and Game Development

In October 1995, I was excited about my future as a game developer. Macromedia had just released Shockwave, and I saw it as a way to develop games that I could distribute myself, over the Web.

Only twice since then have I seen something that has made me just as excited about game development as the original Shockwave. The next time was with the release of Shockwave 3D. And the third time was with the release of ActionScript 3.0.

Flash games have been around for a while, but always as a the younger brother to Shockwave games. Shockwave was faster, more powerful, and eventually in 3D.

With ActionScript 3.0, however, Flash becomes equally as powerful as Shockwave. In some ways, it is more so. For instance, the Flash 9 player is already on 80 percent of the web-browsing computers out there. By the time you get this book, most Flash 8 players would have been upgraded to Flash 9, and we'll be close to 100 percent. Knowing that Flash 9 is almost as ubiquitous as the Web browser itself empowers us as Flash game developers.

Flash 9 even plays on Linux machines. Older versions of Flash play on Web television boxes, game consoles such as the Wii, and even portable devices such as smartphones and the PlayStation Portable. In time, we'll have the Flash 9 player and ActionScript 3.0 on these kinds of devices, too.

You can develop both standalone and web-based versions of your games with Flash. Third-party software enables you to extend your standalone games to make them robust applications.

Flash with ActionScript 3.0 is a great, practical way to make small and medium-size games.

Who Is This Book For?

This book is for anyone using Flash to develop games. However, different types of developers will use this book in different ways.

Someone fairly new to both Flash and programming will be able to use this book as a next step after learning basic programming skills. A motivated fast learner might also be able to use this book to learn ActionScript 3.0 from scratch.

If you have previous experience programming with ActionScript 1.0 or 2.0, you can use this book to get up to speed on ActionScript 3.0.

However, you should try to forget most of what you know about previous versions of ActionScript. Seriously, ActionScript 3.0 is very different from previous versions. In fact, I consider it a whole new programming language.

Many Flash users already know the basics of animation and programming but want to move on to developing games. This is the core audience for the book.

If you are not a programmer at all, but a designer, illustrator, or animator, you can use the examples in this book as a framework for your own games. In other words, you can just swap out the graphics from the source file examples.

Likewise, if you already are an expert ActionScript 3.0 programmer, this book can provide a library of code for you to draw on to make your games. No need to start from scratch.

What Do You Need To Use This Book?

Most readers will need some previous experience with Flash and programming to get the most from this book. You also need the right tools.

Prerequisite Knowledge

Readers should be familiar with the Flash CS3 environment. If you are new to Flash, run through the Flash User Guide that comes with Flash CS3. From inside Flash, choose Help, Flash Help or press F1. You might also want to consider a beginner's book or online tutorial.

This book is not geared toward first-time programmers, unless you are just looking to use the examples by substituting your own graphics. Therefore, you should have some programming experience: ActionScript 1.0, 2.0 or 3.0, JavaScript, Java, Lingo, Perl, PHP, C++, or just about any structured programming language. ActionScript 3.0 is not hard to understand if you are at least somewhat familiar with loops, conditions, and functions. Chapter 1, "Using Flash and ActionScript 3.0," in fact, sums up the basic ActionScript 3.0 syntax.

If you are a programmer, but have never used Flash before, read the parts of the Flash User Guide that pertain to the Flash interface and basic drawing and animation techniques.

Software Applications

You'll need, of course, Flash CS3 Professional or newer. Flash 8 Studio, the previous version of Flash, does not use ActionScript 3.0 and cannot be used with this book.

Flash CS3 is virtually identical on Mac and Windows. The screenshots in this book were taken with the Mac version of Flash, but they should match the Windows version very closely.

Future versions of Flash will most likely continue to use ActionScript 3.0 as the core programming language. Some of the menu choices and keyboard shortcuts might change, but you should still be able to use this book. You might want to consider setting your Publish settings for the Flash 9 player and ActionScript 3.0 to ensure maximum compatibility.

Source Files

You also need the source files for this book. See the end of the Introduction for information about how to obtain them.

Using the Example Games in Your Projects

This book includes 16 complete games, including some gems such as match three, a side-scrolling platform game, and word search. The question I often get is this: "Can I use these games in my project?"

The answer is this: Yes, as long as you modify the games to make them your own, such as changing the artwork, game play, or other content. Posting the games as-is to your Web site is not acceptable. Also, posting the source code or code listings from this book is unacceptable.

When you use these games in your projects, don't try to pass them off as completely your own work. To do so would be unprofessional. Please credit the book with a link to http://flashgameu.com.

However, if you are only using a small portion of the code, or using a game as a basic framework for something very different, no attribution is needed.

Basically, just use common sense and courtesy. Thanks.

What You'll Find in This Book

Chapter 1, "Using Flash and ActionScript 3.0," introduces ActionScript 3.0 and some basic concepts such as game programming strategies and a checklist to help you develop games in Flash CS3.

Chapter 2, "ActionScript Game Elements," presents a series of short code snippets and functions, such as creating text fields, drawing shapes, and playing sounds. This is a useful and practical library of code that we'll be using throughout the book (and you'll be using in your own projects).

Chapters 3 through 12 each contain one or more complete games. The text of the chapter walks you through the game code, enabling you to build it yourself if you want. Or, you could use the source file and walk through the code.

Chapter 3, "Basic Game Framework: A Matching Game," is a little different from the rest of the book. Instead of examining game code for a finished game, it builds a game in ten steps, producing a different Flash movie and source code file with each step. It is a great way to learn how to build Flash games.

Most of the rest of the chapters introduce a special topic before starting a new game. For instance, Chapter 4 starts with an "Arrays and Data Objects" section.

But, the content of this book doesn't stop with the pages in your hands. There is more to be found online.

The FlashGameU.com Website

The **FlashGameU.com** is the companion website to this book. Go there to find the source files, updates, new content, a Flash game development forum, and my blog and podcast on Flash game development.

The source files for this book are organized by chapter, and then further divided into archives for each game. There is a link to download the files at the main page of **FlashGameU.com**.

At **FlashGameU.com**, you'll also find a blog in which I post new content and try to answer reader questions. If you've got a question about something in this book, or about Flash game development in general, you can ask it in the forum or ask me directly through the blog.

Hope to see you there!

This page intentionally left blank

1

Using Flash and ActionScript 3.0

- What Is ActionScript 3.0?
- Creating a Simple ActionScript Program
- Working with Flash CS3
- Writing and Editing ActionScript
- ActionScript Game Programming Strategies
- Basic ActionScript Concepts
- Testing and Debugging
- Publishing Your Game
- ActionScript Game Programming Checklist

ActionScript is a great programming language for making games. It is easy to learn, fast to develop with, and very powerful.

We'll start by looking at ActionScript 3.0 and the Flash CS3 Professional authoring environment. Then, we'll build some very simple programs to get familiar with this new version of ActionScript.

What Is ActionScript 3.0?

ActionScript 3.0 was introduced in 2006 with the release of Flex 2. Flex enables developers to build applications that require the Flash Player, just like Flash does. However, Flash offers a more visual interface for developing applications, one better suited for game development.

ActionScript was introduced in 1996 with the release of Flash 4. It wasn't called ActionScript yet, and you couldn't even type your code. Instead, you chose statements from a series of drop-down menus.

Flash 5 in 2000 improved on that greatly with the formal introduction of ActionScript 1.0. This scripting language contained all the bells and whistles of other web-based development languages, such as Macromedia Director's Lingo, and Sun's Java. But, it came up severely short in speed and power.

Flash MX 2004, also known as Flash 7, brought us ActionScript 2.0, a much more powerful version of the language that made it easier to create object-oriented programs. It was much closer to ECMA Script, a standard for programming languages developed by the European Computer Manufacturers Association. JavaScript, the programming language used in browsers, is also based on ECMA Script.

NOTE

The Flash 9 Player has two separate code interpreters built in to it. The first is for older content and will interpret ActionScript 1.0/2.0 code. The second is a faster code interpreter that works with ActionScript 3.0. You get the best performance out of your games if you stick to only using ActionScript 3.0 code.

ActionScript 3.0 is the culmination of years of development. As each version of Flash came out, developers pushed it to the limit. The next version took into account what developers were using Flash for, and what the weaknesses of the current version of ActionScript were.

Now we have an excellent development environment for 2D game development. You'll find that one of its main strengths is being able to get games up and running with only a small amount of code.

NOTE

Flash CS3 Professional is actually Flash 9. Adobe has simply bundled together versions of various pieces of software—such as Flash, PhotoShop, Illustrator, and Dreamweaver—into their "CS3" package. The technical version number of Flash in CS3 is Flash 9. It is correct to refer to it as either Flash 9 or Flash CS3. The playback engine, which is also used by Flex, is only referred to as the Flash 9 Player.

Creating a Simple ActionScript Program

SOURCE FILES
http://flashgameu.com
A3GPU01_HelloWorld.zip

When introducing a new programming language, it is tradition to start off with Hello World programs. The idea is to simply write a program that does nothing other than display the words *Hello World* on the screen.

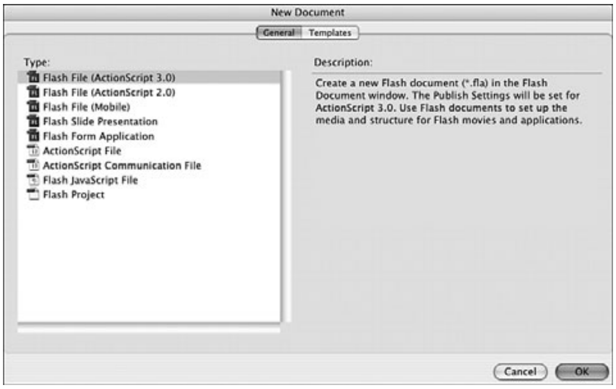
NOTE

The Hello World program dates back to 1974 when it was included in an internal tutorial document at Bell Labs. It was the first program that I learned when I was sat in front of a PDP-11 terminal in school in the late 70s. Just about every beginning programming book has a Hello World example at the beginning.

A Simple Use of `trace`

We can create a limited version of Hello World by using the `trace` function in a script in the main timeline. All that `trace` does is output some text into Flash's Output panel.

To create a new Flash movie, choose File, New from the menu. You'll be presented with the New Document window seen in Figure 1.1.



Create a new Flash document (*.fla) in the Flash Document window. The Publish Settings will be set for ActionScript 3.0. Use Flash documents to set up the media and structure for Flash movies and applications.

Figure 1.1
Choose Flash File (ActionScript 3.0) to create a new Flash movie.

After clicking the OK button, you'll get a new Flash movie named Untitled-1. It will appear as a Flash document window, as shown in Figure 1.2.

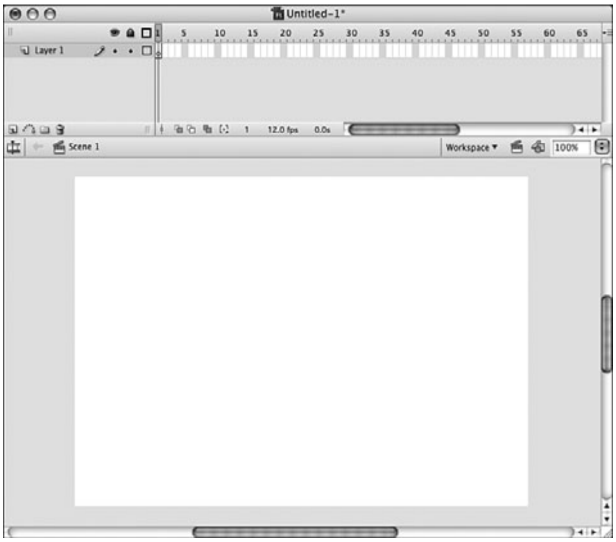


Figure 1.2
The Flash document window includes a timeline and a stage work area.

The top part of the document window includes a timeline, with frames starting with 1 and extending to the right—a little more than 65 frames can be seen in Figure 1.2, although this will depend on the window size. The number of frames can extend as far as an animator needs, but as game programmers, we'll usually only need a few frames to build our games.

The timeline can have one or more layers in it. By default, there is one layer, named Layer 1 in the window.

In Layer 1, you will see a single keyframe, represented by a box with a hollow dot in it under the frame number 1.

NOTE

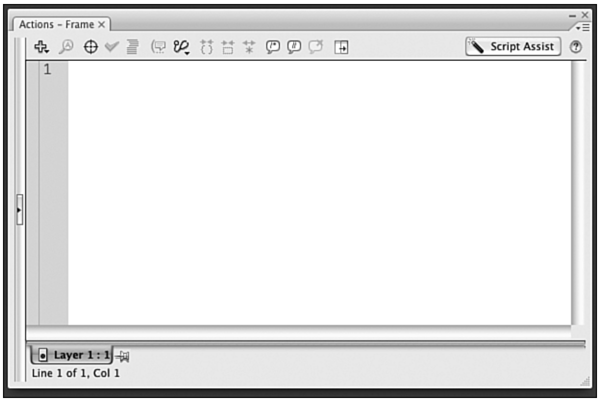
Keyframe is an animation term. If we were learning to animate with Flash, instead of learning to program, we would be using keyframes all the time. Basically, a keyframe is a point in the timeline where the positions of one or more of the animated elements are specifically set. Between keyframes, the elements would change position. For instance, if there were a keyframe on frame 1 where an element is on the left side of the screen, and a keyframe on frame 9 where the same element is on the right side of the screen, in between these keyframes, on frame 5, the element would appear in the middle of the screen.

We won't be using keyframes for animating, but instead we'll be using them to place elements on the screen in different modes: such as "intro", "play", and "gameover".

You can place a script in any keyframe in any layer of the timeline. To do this, select the keyframe, choose the Window menu, and then select Actions.

This brings up the Actions panel. You can see the result in Figure 1.3. It might look different to you because it can be customized in a number of ways, including having a full set of ActionScript commands and functions in a menu on the left side.

Figure 1.3
The Actions panel can also be accessed by the keyboard shortcut Alt+F9 (Windows) or Option+F9 (Mac).



The Actions panel is basically just a text-entry window. However, it can do much more for you, such as help you format your code. We won't be using the Actions panel much for the purposes of this book because most of our code will be in external classes.

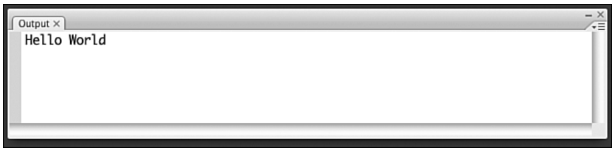
To create this simple Hello World program, enter the following text into the Actions panel:

```
trace("Hello World");
```

That's it. You've created your first ActionScript 3.0 program. To test it, choose Control, Test Movie, or use the shortcut ⌘+Return on Mac, Ctrl+Enter on Windows. If you didn't build the movie yourself, you can open **HelloWorld1.fla** and use this file to test.

Now, look for the Output panel. It will appear, even if you had that panel closed. But, it tends to be a small panel, so it could easily appear in a corner of your screen without you noticing. Figure 1.4 shows what it should look like.

Figure 1.4
The Output panel shows the results of the trace *function call.*



```
Hello World
```

Although this Hello World program technically does output "Hello World," it will only do so while you are testing the movie in Flash 9. If you were to embed this movie in a browser, it would show nothing on the screen. We'll need to do a bit more work to create a real Hello World program.

Creating Screen Output

To have the words *Hello World* display on the screen, we need more than one line of code. In fact, we need three.

The first line will create a new text area to be displayed on the screen, called a *text field.* This is a container to hold text.

The second line will place the words *Hello World* into that text field.

Then, the third line will add that text field to the stage. The stage is the display area of a Flash movie. You can arrange elements on the stage while authoring a movie. During playback, the stage is the area the user sees.

In ActionScript 3.0, creating objects like a text field doesn't add them to the stage. You need to do that yourself. This will come in useful later when you want to group objects together, and not have everything placed directly on the stage.

NOTE
Any visual element in ActionScript 3.0 is called a *display object.* It could be a text field, a graphic element, a button, or even a user interface component (such as a pop-up menu). Display objects can also be collections of other display objects. For instance, a display object can hold all the pieces in a chess game, and the chess board is another display object underneath it. The stage itself is a display object, actually a display object known as a *movie clip.*

Here are the three lines of code for our new Hello World program. These would simply replace the one line of code in frame 1 of the timeline from the preceding example:

```
var myText:TextField = new TextField();
myText.text = "Hello World";
addChild(myText);
```

The code creates a variable named `myText` of the type `TextField`. It then sets the `text` property of this text field to "Hello World" before adding it as a child of the stage display object.

NOTE
The `var` keyword before the first use of the `myText` variable tells the compiler that we'll be creating a variable named `myText`. Then, the colon and the type, `TextField`, tell the compiler what type of value this variable will hold (in this case, a reference to a text field).

The result of this program would be a very tiny "Hello World" in the default serif font at the very upper-left corner of the screen. Choose Control, Test Movie to see it for yourself. The source file is **HelloWorld2.fla**. Figure 1.5 shows this little text field that we have created.

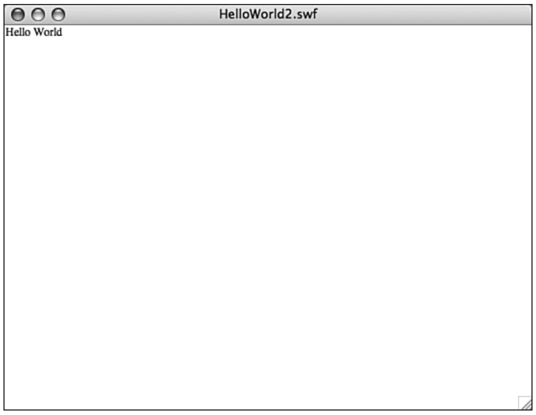


Figure 1.5
The screen shows a tiny "Hello World" at the upper left.

The reason the text appears at the upper left and in that particular font is that we have not set any other properties of the text field. After we learn a little more, we can set the text location, size, and font.

Our First ActionScript 3.0 Class

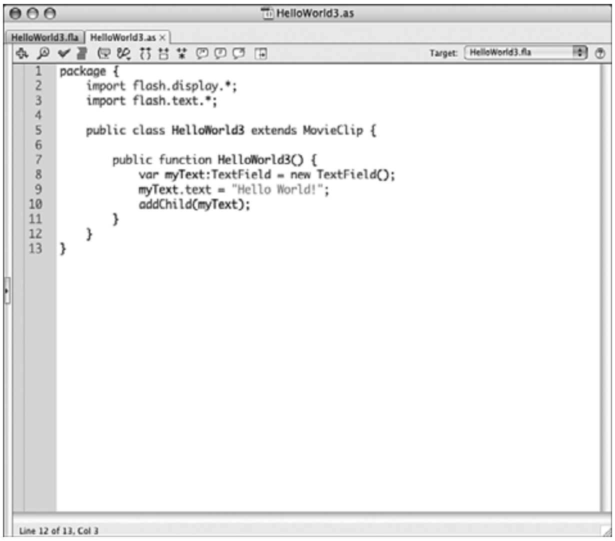
We won't be using scripts in the timeline unless we have something that specifically needs to be done on a certain frame in the timeline. For the most part, our code will exist in external ActionScript class files.

So, let's rebuild the Hello World program as an external class.

NOTE

A class is another way of referring to a Flash object, like a graphic element or the movie itself. We also often refer to a class as the code portion of an object. So you'll have a movie, and the movie's class. This would define what data is associated with the movie, and what functions it can perform.

To make an external ActionScript file, choose File, New, and select ActionScript File. This opens up a new ActionScript document window that occupies the same space as the Flash movie document window. Instead of a timeline and a stage work area, however, we just have a large text editing area, as shown in Figure 1.6.



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

    public class HelloWorld3 extends MovieClip {

        public function HelloWorld3() {
            var myText:TextField = new TextField();
            myText.text = "Hello World!";
            addChild(myText);
        }
    }
}
```

Figure 1.6
The ActionScript document contains a very simple Hello World program.

As you can see in Figure 1.6, this program is much longer than the three-line Hello World program we built earlier. Let's take a look at what each part of the code does.

A class file starts off by declaring that it is a `package` containing a class. Then, it must define what parts of ActionScript are needed in the program. In this case, we need to display objects on the stage, and we need to create a text field. This will require the use of the `flash.display` classes and the `flash.text` classes:

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

NOTE

You'll pretty quickly come to know what library classes you need to import at the start of your programs. These are two out of only a handful that we use throughout this entire book. For more unusual ActionScript functions, you can always look in the Flash 9 Help entry for that function to see which class library to import.

The next line of code is the class definition. In this case, it needs to be a `public` class, which means that it can be accessed by the main movie. The name of the class will be `HelloWorld3`, which must match the name of the file, which is **HelloWorld3.as**.

This class `extends MovieClip`, which means that it will work with a movie clip (in this case, the stage itself):

```
    public class HelloWorld3 extends MovieClip {
```

The class contains a single function. The name of this function is `HelloWorld3`, which exactly matches the name of the class. When a function is named the same as the class name, it will be executed immediately as soon as the class is initialized. This is called the `constructor` function.

In this case, the class is attached to the movie, so this function will run as soon as the movie is initialized.

Inside the function are the same three lines of code we used in the previous example:

```
        public function HelloWorld3() {
            var myText:TextField = new TextField();
            myText.text = "Hello World!";
            addChild(myText);
        }
    }
}
```

To get this code working in a movie, you need to create a fresh new movie. The example is called **HelloWorld3.fla**. This movie doesn't need to have anything in the timeline at all, but it must be assigned a document class. This indicates which ActionScript file will control the movie.

To set a document class, open the Properties panel by choosing Window, Properties, Properties. You'll see the panel shown in Figure 1.7. Then, enter the class name `HelloWorld3` into the document class field.

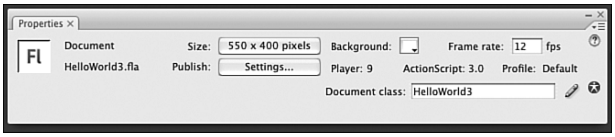

Properties ×
Document HelloWorld3.fla
Size: 550 x 400 pixels
Publish: Settings...
Background:
Player: 9
Frame rate: 12 fps
ActionScript: 3.0
Profile: Default
Document class: HelloWorld3

Figure 1.7
The document class for this movie is set to `HelloWorld3`.

Now the movie knows that it must load and use the **HelloWorld3.as** file. When you test the movie, it will compile the AS class file into the movie. Running the movie will initialize the class, which will run the `HelloWorld3` function and display the "Hello World" text.

Working with Flash CS3

Although most of our work will be in ActionScript, we need to know some terms, and some basics about working with the Flash CS3 timeline, stage, and library.

NOTE

If you are new to Flash, check out "Using Flash" in the Help documentation. This section gives you a detailed explanation of the stage, timeline, library, and other Flash workspace elements and tells you how to handle the Flash interface.

Display Objects and Display Lists

We've already discussed display objects. They are essentially any graphic element. The most versatile of all display objects is the *movie clip,* which is a full graphic element that includes any number of other display objects, plus a timeline for animation.

A simpler version of the movie clip is a *sprite.* A sprite is essentially a movie clip with only one frame. When we create display objects from scratch in ActionScript, we'll usually be making sprites. They are naturally more efficient than movie clips because they don't have the overhead of multiple frames of animation.

Other display objects include things such as text fields, bitmaps, and video.

Some display objects, movie clips, and sprites can have other display objects in them. For instance, you can have a sprite that contains several other sprites, as well as some text fields and bitmaps.

Nesting display objects provides you a way to organize your graphic elements. For instance, you could create a single game sprite to hold all the game elements you create with ActionScript. Then, you could have a background sprite inside of it that contains several background sprite elements. A game pieces sprite could sit on top of that and contain moveable game pieces.

Because movie clips and sprites can contain multiple objects, they will each maintain a list of these items to determine the order in which they are displayed. This is called a *display list.* We can modify this display list to place objects in front of or in back of other objects.

NOTE

The idea of a display list is new to ActionScript 3.0. If you are used to the ActionScript 2.0 method of levels and depths, forget about that and embrace the simpler display list method. With display lists, nothing is at a set level. Instead, the farthest display object is simply first in the list, and the closest is last in the list. You can still move objects around in the list, and the chances of errors or unwanted side effects are greatly reduced.

We can also move display objects from one parent display object to another. This isn't making a copy of the object, but is actually removing it and adding it again. This makes display objects incredibly versatile and easy to work with.

The Stage

The stage is the main graphics work area in Flash. It is a representation of the screen that will be seen by users when they are playing the game.

Figure 1.8 shows the document window with the stage taking up a majority of the space. It also shows the timeline at the top.

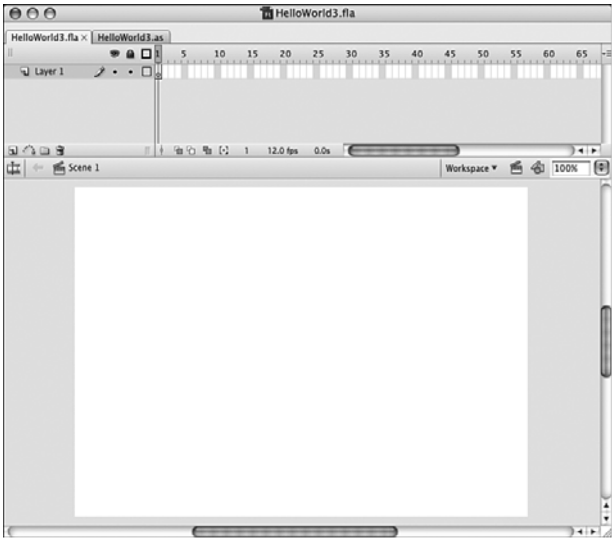


Figure 1.8
The document window includes both the stage and the timeline.

Many of our games will have a completely blank stage and empty timeline. All the graphic elements will be created by the ActionScript code.

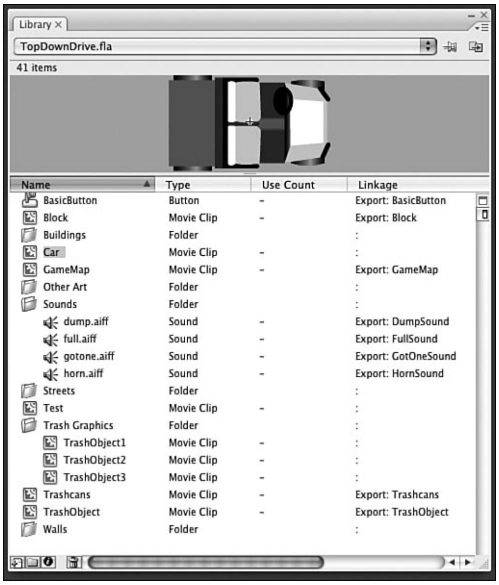
However, many games will have graphic elements already sitting on the stage. This is particularly important when a nonprogrammer graphic designer is involved in making a game. He or she might want to lay out interface elements and adjust them during development. It is simply not practical to have those elements created by ActionScript in cases like this.

During development, the stage can be used as a place to create quick graphic elements. For instance, you can draw using the drawing tools on the stage, select the shape, and then press F8 to create a quick movie clip in the library.

The Library

The Flash library contains any media that you need in your game and will be bundled into the final SWF file. You can also import other media elements into your movie, as you'll see when we import external bitmap images in Chapter 6, "Picture Puzzles: Sliding and Jigsaw."

Figure 1.9 shows the Library panel. It is usually narrower, but I've expanded it to show the Linkage column.


Library

TopDownDrive.fla

41 items

Name	Type	Use Count	Linkage
BasicButton	Button	-	Export: BasicButton
Block	Movie Clip	-	Export: Block
Buildings	Folder		:
Car	Movie Clip	-	:
GameMap	Movie Clip	-	Export: GameMap
Other Art	Folder		:
Sounds	Folder		:
dump.aiff	Sound	-	Export: DumpSound
full.aiff	Sound	-	Export: FullSound
gotone.aiff	Sound	-	Export: GotOneSound
horn.aiff	Sound	-	Export: HornSound
Streets	Folder		:
Test	Movie Clip	-	:
Trash Graphics	Folder		:
TrashObject1	Movie Clip	-	:
TrashObject2	Movie Clip	-	:
TrashObject3	Movie Clip	-	:
Trashcans	Movie Clip	-	Export: Trashcans
TrashObject	Movie Clip	-	Export: TrashObject
Walls	Folder		:

Figure 1.9
The Library panel shows all the media objects enclosed in the current movie.

In Figure 1.9, most of the items in the library are movie clips. The first item is a button, and several that are in the Sounds folder are sounds.

Some of the movie clips have an export name in the Linkage column. These are items that can be pulled from the library by our ActionScript code at runtime.

The Timeline

A Flash movie is broken up into frames. The timeline at the top of the window allows you to choose the frame that is displayed in the stage work area at the bottom of the window. Because we are not producing animations, but game applications, we'll be using the frames to differentiate between different game screens.

Figure 1.10 shows a timeline. Only three frames are in use. They are all keyframes. The first is for a game introduction screen and contains some instructions. The second is the frame where the game is played. The third is a "Game Over" message and a "Play Again" button.

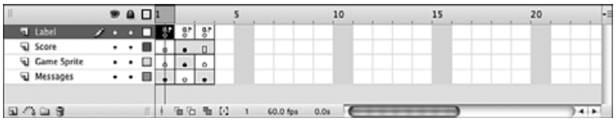
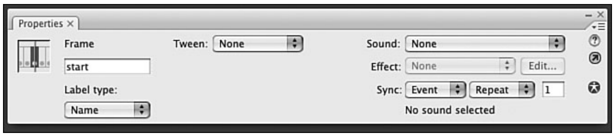


Figure 1.10
The timeline has been expanded slightly using the pull-down menu at the right, so the frames are a little larger.

Each keyframe has a label, although you cannot see it in the timeline. You can see a little flag in the top layer of each frame, which signifies that there is a label there.

To see and set the frame labels, you need to select the frame, and then check the Properties panel. It will contain a Frame field. In this case, it has been set to "start", and you can edit it if you need (see Figure 1.11).


Properties

Frame: start
Label type: Name
Tween: None
Sound: None
Effect: None
Edit...
Sync: Event Repeat 1
No sound selected

Figure 1.11
The Properties panel allows you to set or change the frame label.

If you look back at Figure 1.10, you can see that there are four layers. The first layer, Label, contains three keyframes. The way you create frames and keyframes is to use F5 to add a frame to a layer, and then F7 to add a keyframe among those frames.

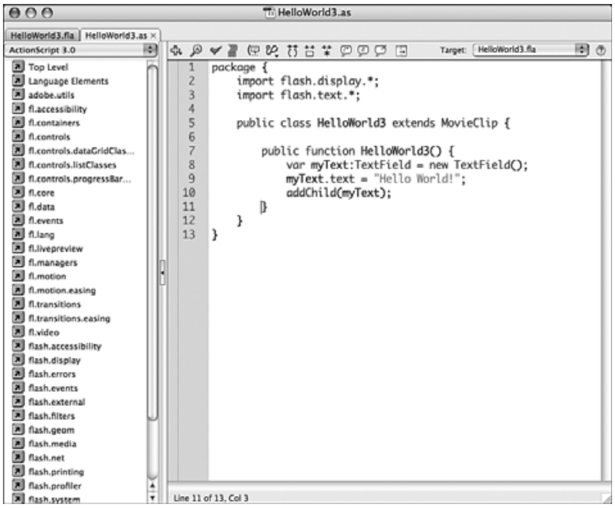
The second layer, named "Score," contains only two keyframes, frame 1 and 2. So, frame 3 is just an extension of frame 2. This means the score elements present during game play on frame 2 is still present on frame 3.

The timeline, stage, and library will be your primary visual tools for developing your games.

Writing and Editing ActionScript

Although it is usually necessary to work in the Flash document somewhat to create a game, we'll be spending most of our time in the ActionScript document window.

We already saw this window in Figure 1.6, but Figure 1.12 shows it differently. On the left is a hierarchical menu of ActionScript 3.0 syntax.



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

    public class HelloWorld3 extends MovieClip {

        public function HelloWorld3() {
            var myText:TextField = new TextField();
            myText.text = "Hello World!";
            addChild(myText);
        }
    }
}
```

Line 11 of 13, Col 3

Figure 1.12
The ActionScript document window features several useful tools at the top.

At the very top of the window, you'll see two tabs. That is because two documents are open: **HelloWorld3.fla** and **HelloWorld3.as**. This allows you to work on the Flash movie and the ActionScript document at the same time. You can switch between them by clicking the tabs. You can also have other ActionScript files open, which proves handy if you are working with multiple ActionScript classes at the same time.

Notice in Figure 1.12 that the lines of code are indented. The proper way to do this is by using the Tab key. When you press Return or Enter at the end of a line of code, the cursor automatically appears indented to the proper level at the next line. If you want to remove a Tab stop to pull a line closer to the left, just press Delete or Shift+Tab.

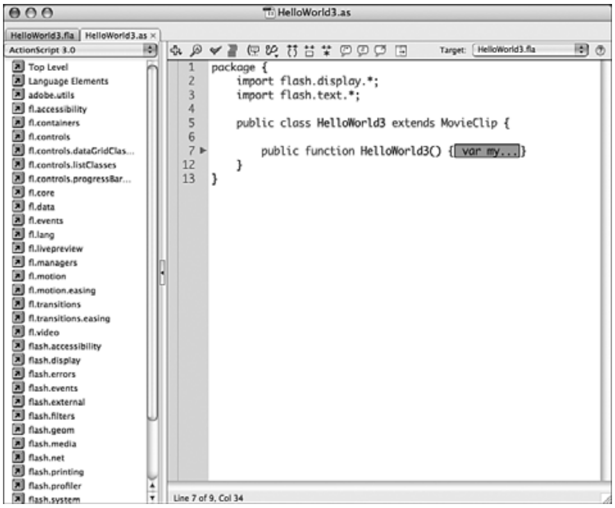
NOTE

You can also select a section of code and press Tab to move it all over to the right by one Tab stop. You can Shift+Tab to move a whole section to the left, too.

The script window tools at the top perform various functions that every ActionScript programmer should know how to use. Here is a list (as shown in the window, from left to right):

- **Add a New Item to the Script.** This is a massive drop-down menu that gives you access to every ActionScript command. There is so much that it is difficult to use for standard commands, but can be useful to find more obscure ones.
- **Find.** Use this to open the Find and Replace dialog box. You can also use ⌘+F (Mac) or Ctrl+F (Windows).
- **Check Syntax.** This is a handy way to have the Flash compiler do a precheck on the syntax of your script. You can see the results in the Output panel.
- **Auto Format.** This takes your entire script and reformats it with consistent tabbing, spacing, and brackets. If you decide to use this, be sure to visit the Preferences for Auto Format to make some decisions about what this button should and should not do.
- **Show Code Hint.** This is probably the most useful of all the buttons. When you start typing a function, such as `gotoAndStop(`, you will get a code hint that instantly appears letting you know what parameters the function accepts. However, if you want to edit the function call later, you can position the cursor inside the function parameters and then use this button to bring back the hints.
- **Debug Options.** This drop-down menu allows you to set and remove breakpoints. We discuss debugging later in this chapter in the "Testing and Debugging" section.

- **Collapse Between Braces.** If you click this button, the current section of code, between curly braces, collapses to a single line. The code is still there, but hidden. You can click on the triangle (Mac) or plus sign (Windows) to the left side of the window, or the Expand All button to expand it. Figure 1.13 shows what it looks like when some code has been collapsed.



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

    public class HelloWorld3 extends MovieClip {

        public function HelloWorld3() {var my...}
    }
}
```

Figure 1.13
A block of code has been collapsed. This is handy when you have a huge script and want to hide sections of code you are not working on.

- **Collapse Selection.** This collapses the code currently selected.
- **Expand All.** This will revert all collapsed sections to their normal status.
- **Apply Block Comment.** Select some code and press this button to turn the selection into a comment by adding `/*` before and `*/` after. See the section "ActionScript Game Programming Strategies" for more about commenting your code.
- **Apply Line Comment.** The current line will be turned into a comment. If multiple lines are selected, all the lines will be turned into comments by adding `//` in front of each line.
- **Remove Comment.** Turns selected comments back into code. This is handy when you want to temporarily remove code from you program. You can comment those lines of code so that they don't compile, and then remove the comment marks to bring the code back.
- **Show/Hide Toolbox.** This button toggles the list of ActionScript at the left side of the window.

At the right of the buttons is a drop-down menu labeled Target. This button enables you to select a Flash movie document that will compile and run when you select Control, Test Movie. This makes it possible to make a change to your code and test the movie without having to switch back to the document window first. Usually, the Flash movie document last viewed is shown here. But, you can select a document if multiple ones are open.

Another important feature of the ActionScript document window is the numbers to the left. Each line has its own number. When you get compiler errors when trying to publish your movie, they will refer to the line number so that you can track down the problem.

ActionScript Game Programming Strategies

ActionScript 3.0 is very versatile. You can follow any number of programming styles and still create games that work well.

However, some programmers prefer certain styles over others. I have chosen a method for this book that allows us to focus on the core game code, perhaps sacrificing some advanced organization.

Single Class Method

The third Hello World program earlier in this chapter is simply one class file linked to a Flash movie of the same name. This simple approach is fast and easy.

NOTE

An alternative is to use different class files for different game objects and processes. This can make it difficult to keep track of what code is where in small games. For instance, if a ball collides with a paddle in a game, is the collision detection in the ball object's class or the paddle object's class?

You are certainly welcome to break the code up into multiple classes if you are familiar with that type of organization from your experience with other programming languages.

With one class file, all our class properties can be clearly defined as variables at the top of the class.

The document class controls the main timeline, meaning that we can call public functions in the class from buttons placed on the stage by designers. We can also easily control the main timeline, jumping to different frames.

Smallest Step Approach

This next piece of information might be the most important in the book. It is simply this:

If you can't figure out how to program something, break it into smaller steps until you can.

Beginner programmers, and some experienced programmers who simply forget this rule, often get stuck while writing code. They think: “I don’t know how to make the program do a certain task.”

However, this is simply a case of the task actually being several tasks, not just one.

For example, a programmer might want to make a spaceship rotate around when the player presses the arrow keys. The programmer gets frustrated because he or she is not sure how to accomplish this task.

The key is to break up the task of “rotating a spaceship”: Check to see whether the left arrow is pressed. Subtract from the `rotation` property of the ship sprite. Check to see whether the right arrow is pressed. Add to the `rotation` property of the ship sprite.

The task of rotating a spaceship is actually four smaller tasks combined into one.

Sometimes beginning programmers make the same mistake in a bigger way. They assume they cannot create an entire game, because it seems to complex. But, if you break the game into smaller and smaller tasks (and take each one step at a time), you can create any game.

A simple whack-a-mole game might require fewer than a hundred tasks, whereas a complex platform game may require several hundred. But each task, if broken into its simplest steps, is just as easy to build.

Good Programming Practices

While learning to use ActionScript 3.0 to make games, it is also a good idea to keep some general good programming practices in mind. These aren’t so much rules, as guidelines. Even I break them here in there in the very pages of this book. But, there is no doubt that you would be a better programmer if you learn about these practices.

Use Comments Well

Comment your code with meaningful but simple comments.

What seems like extra work now will have you thanking yourself months from now when you need to go back and modify your code.

If you are working with other programmers, or think there is even a remote chance that someone else will have to modify your code at some point in the future, this guideline should become a rule.

There are generally two types of comments: *line comments* and *block comments*. A line comment is simply a short phrase at the end of a line, or sometimes a single line comment just before the line of code. A block comment is a larger comment, usually one sentence or more, before a function or a section of code:

```
someActionScriptCode(); // this is a line comment

// this is a line comment
someActionScriptCode();

/* This is a block comment.
Block comments can be much longer.
And contain a description of what is to come. */
```

It is equally as important to make your comments meaningful and brief. Don't simply restate what the code already says, like this:

```
// loop 10 times
for (var i:int=0;i<10;i++) {
```

Also, don't use a paragraph of text when a few words will do. A long and wordy comment can be just as useless as no comment at all. Don't overdo it.

Use Descriptive Variable and Function Names

Don't be afraid to use long and descriptive names for variables and functions. If you do, the code will become partially self-explanatory. Here is an example:

```
public function putStuff() {
     for(var i:int=0;i<10;i++) {
          var a:Thing = new Thing();
          a.x = i*10;
          a.y = 300;
          addChild(a);
     }
}
```

What does that code do? It looks like it places copies of a movie clip on the screen. But what movie clip, for what purpose? How about this:

```
public function placeEnemyCharacters() {
     for(var enemyNum:int=0; enemyNum<10; enemyNum++) {
          var enemy:EnemyCharacter = new EnemyCharacter();
          enemy.x = enemyNum*10;
          enemy.y = 300;
          addChild(enemy);
     }
}
```

Returning to this code months later will be much easier.

NOTE

One common exception to this is the use of `i` as the incremental variable used with `for` loops. In the preceding example, I would have left the `i` and not changed it to `enemyNum`. Either way is fine, but it has become pretty standard for programmers to use `i` in `for` loops. In fact, nested `for` loops usually go on to use `j` and `k` in the inner loops, too.

Turn Repetitive or Similar Code into Functions

If you need to use the same line of code more than one time in a program, consider turning it into a function of its own and call that function instead.

For instance, there might be several places in your game where you want to update the score. If the score is displayed in a text field named `scoreDisplay`, you do it like this:

```
scoreDisplay.text = "Score: "+playerScore;
```

But instead of including this same line of code in five places, you should put a function call in the five places instead:

```
showScore();
```

Then, the function can look like this:

```
public function showScore() {
     scoreDisplay.text = "Score: "+playerScore;
}
```

With this code only in one place, it is trivial to change the display word *Score* to *Points*. You don't need to search and replace throughout your code, because it is only in one spot.

You can do the same even if the code isn't identical. For instance, suppose you have a loop where you place 10 copies of movie clip A on the left side of the stage, and another loop where you place 10 copies of movie clip B on the right side of the stage. You could create a function that takes the movie clip reference and the horizontal position for placement, and will place the movie clips. Then, you can call it twice, once for movie clip A and once for movie clip B.

Test Your Code in Small Pieces

As you write your code, test it in as small pieces as possible. This way, you catch errors as you write your code.

For instance, if you want to make a loop that places ten circles on the screen at random locations with random colors, you want to first create the ten circles at random locations. Test it, and get that portion working just like you want. Then, add the random color functionality.

This basically is an extension of the "smallest step approach." Break your programming tasks into small steps. Create the code for each step. Then, test each step.

Basic ActionScript Concepts

Let's take a look at the most basic programming syntax in ActionScript 3.0. If you are new to ActionScript, but have been using another programming language, this will be a quick way to see how ActionScript works.

In case you have used ActionScript or ActionScript 2.0 before, I point out some places where ActionScript 3.0 differs.

Creating and Using Variables

Storing values in ActionScript 3.0 can be done with a simple assignment statement. However, you need to declare a variable the first time you use it. You can do this by placing `var` before the first use of the variable:

```
var myValue = 3;
```

Alternatively, you could declare the variable first and use it later:

```
var myValue;
```

NOTE

In ActionScript 2.0, the `var` statement is not required. In ActionScript 3.0, it is.

When you create a variable in this way, it is a very versatile `Object` type. This means it can hold any type of variable value: A number, a string such as "Hello", or something more complex like an array or movie clip reference.

However, if you declare a variable to be of a specific type, you can only use the variable to store values of that same type:

```
var myValue:int = 7;
```

An `int` variable type can be any integer, positive or negative. A `uint` variable is only for positive integers. If you want to use fractional values, also known as floating-point numbers, you must use the `Number` type:

```
var myValue:Number = 7.8;
```

There are also `String` and `Boolean` types. Strings hold text, and Boolean values must be either `true` or `false`.

These are the basic primitive types. However, you can also have arrays, movie clip and sprite references, and new types that match the code classes you create.

NOTE

There is a definite efficiency advantage to using narrowly defined variables. For instance, `int` values can be accessed many times faster than `Number` values. This can help you speed up critical game processes if you stick to as basic of a type as possible for all variables.

Operations on numeric variables are like almost any other programming language. Addition, subtraction, multiplication, and division are performed with the `+`, `-`, `*`, and `/` operators:

```
var myNumber:Number = 7.8+2;
var myOtherNumber:int = 5-6;
var myOtherNumber:Number = myNumber*3;
var myNextNumber:Number = myNumber/myOtherNumber;
```

You can also use special operators to simplify operations. For instance, the `++` operator will increment a variable by one. The `--` operator decreases it by one:

```
myNumber++;
```

You can use `+=`, `-=`, `*=`, and `/=` to perform an operation on the original variable. For instance, this will add seven to the variable:

```
myNumber += 7;
```

You can also use parenthesis to set the order of operations:

```
var myNumber:Number = (3+7)*2;
```

Strings can also be manipulated with the `+` operator and the `+=` operator:

```
var myString:String = "Hello";
var myOtherString = myString+"World";
myString += "World";
```

When we use variables in classes, they become properties of that class. In that case, we must define them further as either `private` or `public`. The difference is that `private` variables cannot be accessed by code outside of the class. For most purposes, this is what you want as the class functions should be the only things that can alter the class variable values.

Condition Statements

The `if` statement in ActionScript is the same as it is in many programming languages:

```
if (myValue == 1) {
    doSomething();
}
```

The `==` comparison checks for general equality. You can also use `>`, `<`, `>=`, and `<=` for greater than, less than, greater than or equal to and less than or equal to, respectively.

You can add `else` and `else if` to extend the `if` structure:

```
if (myValue == 1) {
     doSomething();
} else if (myValue == 2) {
     doSomethingElse();
} else {
     doNothing();
}
```

You can also have more complex conditions with `&&` and `¦¦`. These represent the `and` and `or` comparison operators.

NOTE
Before ActionScript 3.0, you could use the keywords and and or as alternatives to && and | |. These are no longer accepted.

```
if ((myValue == 1) && (myString == "This")) {
     doSomething();
}
```

Loops

Looping is done with the `for` statement or the `while` statement.

The `for` statement has three parts: the initial statement, a condition, and a change statement. For instance, the following code sets the variable `i` to zero, loops as long as it is less than ten, and increases the value of `i` each time through the loop:

```
for(var i:int=0;i<10;i++) {
     doSomething();
}
```

You can use the `break` command to exit a loop at any time. The `continue` command skips the rest of the lines of code in the loop and begins the next iteration through the loop.

A `while` loop is basically a loop that continues forever as long as an initial condition is met:

```
var i:int = 0;
while (i < 10) {
     i++;
}
```

A variation of the `while` loop is the `do` loop. It is essentially the same, except the conditional statement is after the loop, ensuring that it executes at least once:

```
var i:int = 0;
do {
    i++;
} while (i <10);
```

Functions

To create functions in ActionScript 3.0, you just need to declare the function, what parameters it takes, and what it returns. Then define the function with the code inside it.

If the function is in a class, you need to also define whether it is a public or private function. A private function cannot be accessed outside of the class. With our single-class game development method, we'll be using mostly private classes.

NOTE

You might find that functions are sometimes referred to as *methods*. In the documentation, the term *method* is used frequently, but the keyword `function` is used to define it, as you will see below. So, I prefer to use the term *function*.

Here is a simple function from inside a class. If this function were in the main timeline, rather than in a class, we would leave the `private` keyword off:

```
private function myFunction(myNumber:Number, myString:String): Boolean {
    if (myNumber == 7) return true;
    if (myString.length < 3) return true;
        return false;
}
```

All this example function does is to return `true` if either the number is seven, or the string is less than three. It is just a simple example to show the syntax behind creating a function.

Testing and Debugging

No one, not even the most experienced programmer, writes perfect code. So, we must write code, test, and debug.

Types of Bugs

There are three reasons to debug your code. The first is that you get an error message when it compiles or runs. In this case, you must figure out the problem and correct it. Usually, you can see the problem immediately. For example, a misspelled variable name.

The second reason is that the program does not work as expected. Perhaps a character is supposed to move, but doesn't. Or, user input is not accepted. Or, maybe the bullets the hero fires at the enemy pass right through them. This type of bug needs to be hunted down, and it can sometimes take a while.

NOTE

By far the most common type of question I get from other programmers is that they have some code that doesn't work as expected. Can I tell them what is wrong with it?

Yes, but the answer is right there in front of them; they just need to use their debugging skills to find it. And, as the creator of the code, they are usually in a much better position to do that.

A third reason to debug your code is to improve it. You can track down inefficiencies and problems that cause slowdowns. Sometimes this is just as critical as a bug, because a slow game might be unplayable.

Methods of Testing

You can track down issues with your code in several ways. The simplest is to just walk through the code in your head. For instance, walk though the following code, line by line, and do the calculations like you were the computer:

```
var myNumber:int = 7;
myNumber += 3;
myNumber *= 2;
myNumber++;
```

You don't need to run the code to tell that the value of `myNumber` is now 21.

For situations where the code is too long, or the calculations are too difficult, a simple `trace` command sends information to the Output panel for you to examine:

```
var myNumber:int = 7;
myNumber += 3;
myNumber *= 2;
myNumber++;
trace("myNumber = ", myNumber);
```

NOTE

Before Flash CS3, the `trace` command would only accept a single string and send it to the Output panel. Now it can accept any data type, as well as multiple items separated by commas. This makes it much more useful as a debugging tool.

I use `trace` statements often while developing. For instance, if the player makes a bunch of choices at the start of the game, I'll send the results of those choices to the Output panel with `trace`. That way, while I'm testing, I have a reminder of what options I chose before playing the game in case something unexpected happens.

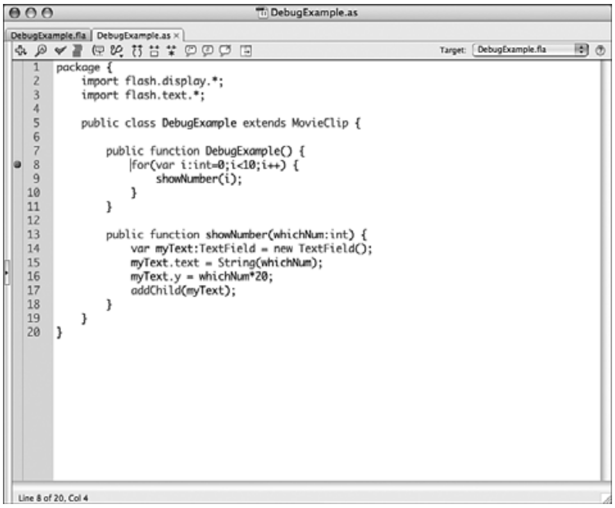
Using the Debugger

With Flash CS3, you can use a runtime debugger to check your code while your movie is running. If you have used the ActionScript 2.0 debugger, keep in mind that the ActionScript 3.0 debugger is a whole new animal.

Setting a Breakpoint

The simplest way to debug a program is to set a break point. You can do this by selecting a line of your code and choosing Debug, Toggle Breakpoint from the menu. You can also press +B (Mac) or Ctrl+B (Windows) to set or remove a breakpoint.

Figure 1.14 shows the **DebugExample.as** code, with a breakpoint set. You can see it as a dot on the left side of the window before the eighth line. The program simple creates 10 text fields with the numbers 0 through 9 in them, and places them vertically down the left side of the screen.



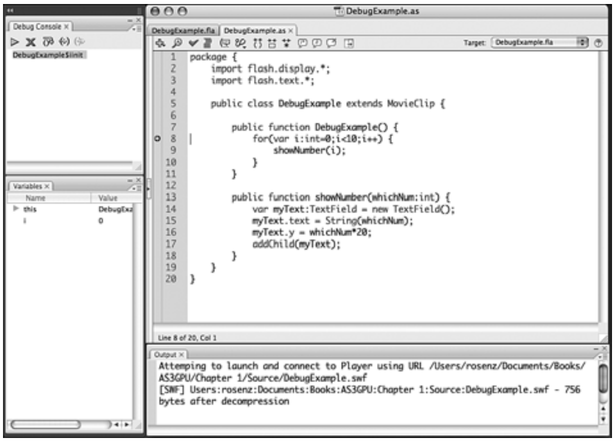
```
1  package {
2      import flash.display.*;
3      import flash.text.*;
4  
5      public class DebugExample extends MovieClip {
6  
7          public function DebugExample() {
8 ●            for(var i:int=0;i<10;i++) {
9                  showNumber(i);
10             }
11         }
12 
13         public function showNumber(whichNum:int) {
14             var myText:TextField = new TextField();
15             myText.text = String(whichNum);
16             myText.y = whichNum*20;
17             addChild(myText);
18         }
19     }
20 }
```

Line 8 of 20, Col 4

Figure 1.14
The cursor was placed in line 8, and then Debug, Toggle Breakpoint was chosen to set a breakpoint there.

After a breakpoint is set, you can use Debug, Debug Movie rather than Control, Test Movie to test the movie. When the program hits the line with the breakpoint, it will pause and display a variety of information to you in various debug windows.

With the breakpoint set at line 8, and if you use Debug, Debug Movie, a whole set of debugging panels show up, in addition to the running Flash movie. You can see them all in Figure 1.15.



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

    public class DebugExample extends MovieClip {

        public function DebugExample() {
            for(var i:int=0;i<10;i++) {
                showNumber(i);
            }
        }

        public function showNumber(whichNum:int) {
            var myText:TextField = new TextField();
            myText.text = String(whichNum);
            myText.y = whichNum*20;
            addChild(myText);
        }
    }
}
```

```
Attemping to launch and connect to Player using URL /Users/rosenz/Documents/Books/
AS3GPU/Chapter 1/Source/DebugExample.swf
[SWF] Users:rosenz:Documents:Books:AS3GPU:Chapter 1:Source:DebugExample.swf - 756
bytes after decompression
```

Figure 1.15
The debug panels show you various information about the state of your program.

Stepping Through the Code

There are five buttons at the top of your Debug Console panel at the upper left. The first is a Continue button, which will continue the movie from the point at which it is stopped. The second is an X. This will end the debugging session and continue the movie from this point on without debugging.

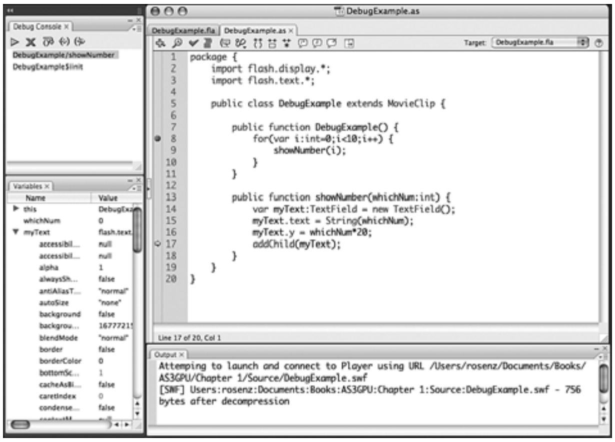
The other three involve stepping through the code. The first will execute the current line and move on to the next. If the current line of code calls another function, it will run that function. On the other hand, the next button, Step In, steps the program into a new function if one exists on the same line. Using this button again and again will mean you visit every individual line of the program, instead of skipping over function calls.

The last button will step out of the current function. So, use this button to finish the current function and go to the next line of the function you just left.

Figure 1.16 shows the debug panels after you step into the `showNumber` function and then move down a few lines. You can see that the Variables panel shows the value of `i`. You can also expand the `myText` variable to see all the properties of the text field.

At the upper left, you can see where you are in the program. You are currently in the `showNumber` function, which was called from the constructor function for the class. This comes in handy when you have a function that could be called form multiple places.

Knowing how to use the debugger to fix bugs and unexpected behavior is just as important as knowing how to write the code in the first place. As you work on the games in this book and try to modify them to fit your needs, also work on learning how to debug.



```
DebugExample/showNumber
DebugExample$iinit
```

Name	Value
▶ this	DebugExa
whichNum	0
▼ myText	flash.text.
accessibil...	null
accessibil...	null
alpha	1
alwaysSh...	false
antiAliasT...	"normal"
autoSize	"none"
background	false
backgrou...	1677721
blendMode	"normal"
border	false
borderColor	0
bottomSc...	1
cacheAsBi...	false
caretIndex	0
condense...	false

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

    public class DebugExample extends MovieClip {

        public function DebugExample() {
            for(var i:int=0;i<10;i++) {
                showNumber(i);
            }
        }

        public function showNumber(whichNum:int) {
            var myText:TextField = new TextField();
            myText.text = String(whichNum);
            myText.y = whichNum*20;
            addChild(myText);
        }
    }
}
```

```
Attemping to launch and connect to Player using URL /Users/rosenz/Documents/Books/
AS3GPU/Chapter 1/Source/DebugExample.swf
[SWF] Users:rosenz:Documents:Books:AS3GPU:Chapter 1:Source:DebugExample.swf - 756
bytes after decompression
```

Figure 1.16
The debug panels show the progress of the program as you step through it.

Publishing Your Game

After you have completed a game and tested it to your satisfaction, it is time to publish it. Flash games are usually published to the Web by embedding them in HTML pages.

Flash CS3 makes this relatively easy, but there are some options you should understand before publishing.

You can access the Publish Settings dialog by choosing File, Publish Settings. There are typically three sections in the Publish Settings dialog: Formats, Flash, and HTML.

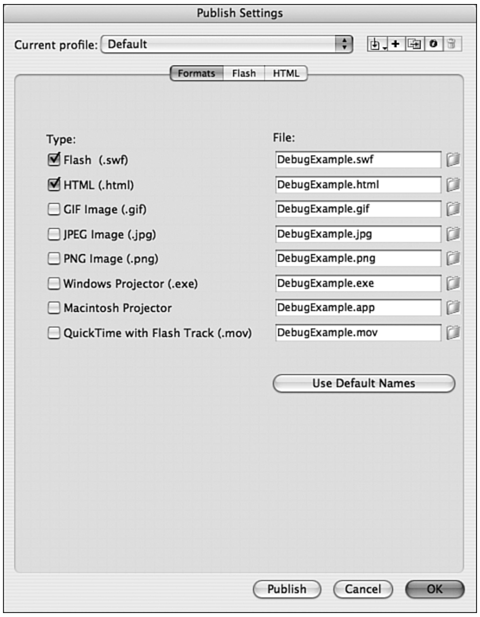
Formats

The Formats settings, shown in Figure 1.17, allow you to select which files to export.

The image formats are mostly for substitutions when the user doesn't have the Flash player installed. And, the QuickTime export is for placing a Flash 5-era Flash movie inside a QuickTime file. None of these apply to us as ActionScript 3.0 game developers.

The two Projector options can be used to make stand-alone versions of our games. These represent a whole different way to export your finished movies.

Figure 1.17
Only Flash and HTML formats are selected for export.



Type:	File:
✓ Flash (.swf)	DebugExample.swf
✓ HTML (.html)	DebugExample.html
☐ GIF Image (.gif)	DebugExample.gif
☐ JPEG Image (.jpg)	DebugExample.jpg
☐ PNG Image (.png)	DebugExample.png
☐ Windows Projector (.exe)	DebugExample.exe
☐ Macintosh Projector	DebugExample.app
☐ QuickTime with Flash Track (.mov)	DebugExample.mov

If you have a custom web page template that you already use on your site, the HTML option might not be necessary. In that case, you don't want a default page to embed your game into. However, you might want to export it anyway and then take the body code from this sample page to use in the proper space on your own page.

Flash

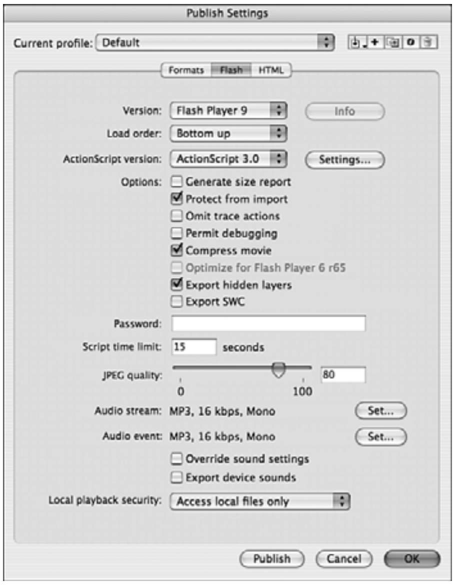
The Flash settings are the most critical to exporting a complex Flash movie like our games. You'll want to set it to export Flash Player 9 files, with ActionScript version set to ActionScript 3.0 (see Figure 1.18).

You'll also want to set the Protect from Import option. This will make it harder for someone to download your movie and modify it for his own use.

NOTE

Unfortunately, there is no surefire way to protect your Flash movie after it has been put on the Web. There are decompiler programs out there that will take a compressed and protected SWF file and convert it to a usable FLA movie. Using Protect from Import and Compress Movie will make this more difficult, but it is always a danger.

The rest of the Flash settings concern compression and security settings. You can check the Flash documentation for detailed descriptions of each.



Publish Settings

Current profile: Default

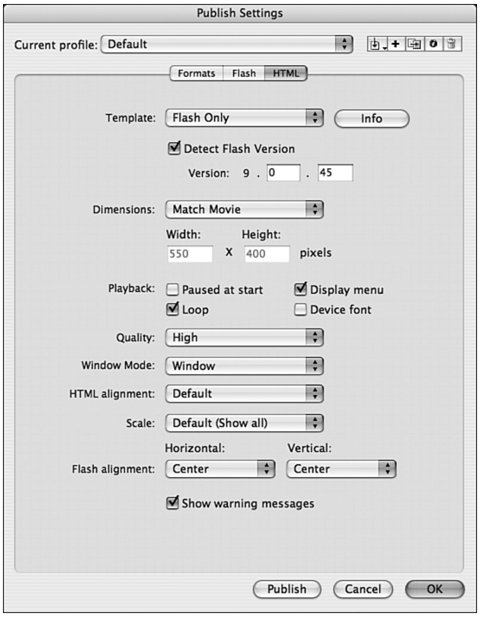
Formats | Flash | HTML

Version: Flash Player 9
Load order: Bottom up
ActionScript version: ActionScript 3.0
Options:
☐ Generate size report
☑ Protect from import
☐ Omit trace actions
☐ Permit debugging
☑ Compress movie
☐ Optimize for Flash Player 6 r65
☑ Export hidden layers
☐ Export SWC
Password:
Script time limit: 15 seconds
JPEG quality: 80
Audio stream: MP3, 16 kbps, Mono
Audio event: MP3, 16 kbps, Mono
☐ Override sound settings
☐ Export device sounds
Local playback security: Access local files only

Figure 1.18
These are a good set of settings for general Flash game use.

HTML

The HTML settings are only important if you want to use the HTML page created by Publishing on your site. But, it is at least a good idea to see how Adobe thinks you should publish your Flash movies. Figure 1.19 shows these options.



Publish Settings

Current profile: Default

Formats | Flash | HTML

Template: Flash Only
☑ Detect Flash Version
Version: 9 . 0 . 45
Dimensions: Match Movie
Width: 550 X Height: 400 pixels
Playback:
☐ Paused at start
☑ Display menu
☑ Loop
☐ Device font
Quality: High
Window Mode: Window
HTML alignment: Default
Scale: Default (Show all)
Flash alignment: Horizontal: Center Vertical: Center
☑ Show warning messages

Figure 1.19
The HTML settings let you choose an HTML template to export with the Flash movie.

The default setting of Flash Only uses JavaScript to embed the movie. It relies on the file **AC_RunActiveContent.js,** which is produced when publishing, too. The main HTML page then loads the JavaScript in this file, which then places the Flash movie in a <div> tag inside the web page.

NOTE

Why go to all the trouble of using JavaScript when you can just use a simple <object>/<embed> tag, like you might have done in years past? Because of a patent dispute, Microsoft had to change the way media is embedded into pages in Internet Explorer. Any media embedded directly into a page now requires a click to activate it in Internet Explorer 7 and some versions of Internet Explorer 6. However, this JavaScript method avoids that extra click.

One popular option is to have the Flash movie scale to fill the entire browser window. This can be done by simply changing the Dimensions setting to Percent and the Width and Height to 100. Now the movie will scale as large as it can, while still fitting in the window and maintaining its proportions.

Changing the Scale to Exact Fit will allow it to lose its proportions and scale vertically to fit the window height and horizontally to fit the window width.

Because all the vector graphics in Flash scale nicely, and your code will work fine at any scale, it is sometimes a good idea to allow the player to adjust the size of the game by simply changing the size of their browser window. This way, people with small monitors and large monitors can play the games as they like.

Another option is the Quality setting. At the default High setting, the Flash player renders the image at high resolution to get the best anti-aliasing effects at the edges of vector shapes. Changing it to Medium lessens the resolution of anti-aliasing, but increases the performance of the movie. Using Auto High means that it will try to use the High setting, but drop down to Medium if playback is too slow. Using Low quality will remove all anti-aliasing, but offers the highest playback speed.

ActionScript Game Programming Checklist

When you are building a Flash game, there are a lot of factors to consider. Sometime it is easy to forget a key element which leads to the game not working correctly. To help you avoid some simple problems, here is a handy checklist that you can refer to.

Publishing and Document Settings

It is easy to forget that there are critical settings in the Publish Settings dialog and the movie's Properties panel.

Is the Document Class Set Correctly?

Figure 1.7 showed how to set the document class using the movie's document panel. Forgetting to set this means that the movie will run and simply ignore the class that you created.

Publish Settings Set Correctly?

Make sure you set the Publish Settings so that the Flash movie is compiled for Flash 9 and ActionScript 3.0. It is doubtful that your movie will even compile if these are not set correctly, but it might be possible.

Check Security Settings

In the Publish Settings, in the Flash section, there is a Local Playback Security setting. It can be set to either Access Local Files Only or to Access Network Files Only. To make sure Flash movies are secure, you have to choose one or the other.

This can be a problem if you have local files you need to access, and this is set to Access Network Files Only. If you are using external files at all, and things are not working as expected when you upload to a server, make this the first place you check.

Class, Function, and Variable Names

Even if you try to follow the good programming practices noted earlier in this chapter, you can still make some simple mistakes that could be hard to track down.

Remember Case Sensitivity

When you name a variable or function, case matters. So, `myVariable` and `myvariable` are completely different. Likewise, a class named `myClass` will run the function `myClass` when it initializes. If you have named it `myclass` by accident, however, it will not be called.

Differences in variable names will usually be caught by the compiler because a misspelled variable name would not have been initialized. But, it is possible to forget that you have declared a variable and declare it again, with different capitalization. This is just something to be on the watch for.

Are Movie Clip Class Files Present?

If a movie clip is given Linkage properties to be used by ActionScript, it can use either the default dynamic class, or you can create a class for it. For instance, you can make an `EnemyCharacter` movie clip and then have an **EnemyCharacter.as** class file that is tied to it.

However, it is easy to forget this class or misname it. For instance, an **Enemycharacter.as** file will simply be ignored, and not attached to the `EnemyCharacter` movie clip.

Do Classes Extend the Correct Type?

You can start off a movie's class with a definition like this:

```
public class myClass extends Sprite {
```

However, by extending a `Sprite` rather than a `MovieClip`, you are assuming that the movie only has one frame. Any code that refers to other frames won't work as expected.

Is the Constructor Function Set to the Right Name?

If you have a class named `myClass`, the constructor function should be named exactly `myClass`; otherwise, it will not run when the class is initialized. Alternatively, if you don't want it to run right away, name it something like `startMyClass` and call it after the frame starts.

Runtime Issues

There are also problems that don't cause compiler errors, and don't appear to be problems at all at first. Then, they can show up later in development and be very frustrating to track down.

Are You Setting Properties before the Object Is Ready?

This one drives me crazy. Basically, what happens is that you jump to a new frame in the movie or a movie clip, and then try to set or access a property of an object there. However, the frame and its objects haven't been initialized yet, so the property doesn't exist.

TooEarlyExample.fla and **TooEarlyExample.as** illustrate this. The class jumps the main timeline to frame 2, where two text fields await. It immediately tries to set the text of the first field, but that just calls a runtime error message. The second field is set when the movie is done initializing and runs the script in that frame. That script in turn calls a function in the class. This function sets the text of the second field without a problem.

Are You Disposing of Objects?

Although this might not cause much of a problem, it is a good practice to remember to dispose of all the objects you've created after you are done using them. For instance, if you have the player shoot bullets around the screen, they might be able to hold down a key and shoot thousands in a minute. When they leave the visible area of the screen, you don't want these to hang around in memory and get tracked.

To completely remove an object, you just need to get rid of all references to it in your variables and arrays, and use `removeChild` to take it out of its display list.

Are All Variables Well Typed?

Another factor that might not cause immediate problems, but might still be a long-term issue, is variable typing. Don't use the `Number` type when `int` or even `uint` will do. The later are much faster and take up less memory. If you have thousands of numbers stored in arrays, you might see some slowdown by using `Number` when `int` would do.

Worse than this example is using untyped variables, which are `Objects`. They can store numbers and integers, but have far more overhead. Also, look out for creating `MovieClips` that can be single-framed `Sprites`.

Testing Issues

These items relate to things that can happen during testing or things that should be part of your testing approach.

Do You Need to Disable Keyboard Shortcuts?

If you are using keyboard input while testing your movies, you might find that some keys don't respond. This is because the test environment has some keyboard shortcuts that are taking these key presses.

To turn the keyboard shortcuts off in the testing environment, and allow your movie to act like it would when on the Web, choose Control, Disable Keyboard Shortcuts.

Have You Tested at Other Frame Rates?

If you are using time-based animation, it shouldn't matter what frame rate you are set to, 1 or 60; the animation should move along at the same speed. However, it is worth testing at a low frame rate, say 6 or 12, to see what users on slow machines might see. We'll use time-based animation throughout this book.

Also, it is worth testing at slow and high frame rates to see whether there isn't some system that is still using time-based animation or responses.

Have You Tested from a Server?

A similar problem rears its head when you assume that objects are all present at the start of a movie. The truth is that Flash movies stream, which means they begin playing before all the media has been loaded.

When you are testing a movie locally, however, all the media is instantly there. Then, when you upload and test on a server, some of it might be missing for the first few seconds or even minutes.

NOTE

When you test a movie, you can restart the test by selecting View, Simulate Download. Also, look at View, Download Settings to set a desired simulated download rate, like 56K. Then, the movie will restart with the objects streamed in at the desired rate. But, I'd also test with a live server to be sure.

The solution to any problem that might arise is to have a loading screen that does nothing but wait for all media to be streamed. We'll look at an example of a loading screen in Chapter 2, "ActionScript Game Elements."

This checklist should make it easier for you to avoid common problems and devote more time to creating games, and less time to tracking down bugs.

Now that we have the basics of ActionScript 3.0 covered, the next chapter looks at some short examples of building blocks that you can use to make games.

2

ActionScript Game Elements

- Creating Visual Objects
- Accepting Player Input
- Creating Animation
- Programming User Interaction
- Accessing External Data
- Miscellaneous Game Elements

Before we build complete games, we'll start with some smaller pieces. The short programs in this chapter give us a look at some basic ActionScript 3.0 concepts and also provide us with some building blocks to use later, and in your own games.

SOURCE FILES
http://flashgameu.com
A3GPU02_GameElements.zip

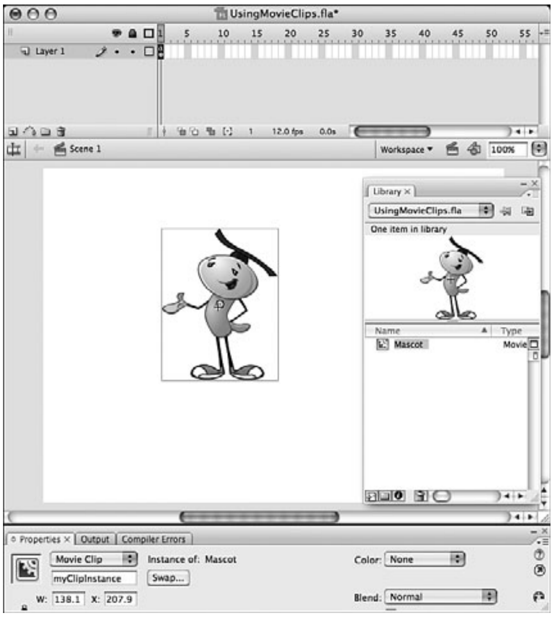
Creating Visual Objects

Our first few elements involve the creation and manipulation of objects on the screen. We'll pull some movie clips from the library, turn movie clips into buttons, draw some shapes and text, and then learn to group items together in sprites.

Using Movie Clips

When you've got a movie clip in the library and want to bring it into your game, there are two ways to do it.

The first way is to drag and drop the movie clip onto the stage and give it an instance name in the Property Inspector. Figure 2.1 shows a movie clip moved from the library to the stage, and then named `myClipInstance` in the Property Inspector.



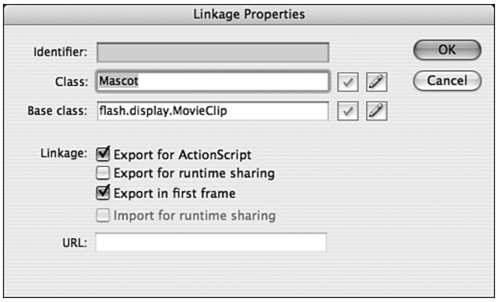
Movie Clip
Instance of: Mascot
myClipInstance
W: 138.1 X: 207.9
Color: None
Blend: Normal

Figure 2.1
The movie clip object is named `Mascot` *in the library, but the instance of the movie clip on the stage is named* `myClipInstance`*.*

Then, you can manipulate the properties of the movie clip by referring to it by name, like this:

```
myClipInstance.x = 300;
myClipInstance.y = 200;
```

The second way to bring a movie clip into your game uses purely ActionScript code. But first, you must set the movie clip to be accessible by setting its Linkage properties. Select the symbol in the library and use the Library panel's drop-down menu to select Linkage. Turn on Export for ActionScript and set the class name. You'll get a dialog box that looks like the one shown in Figure 2.2.


Linkage Properties

Identifier:
Class: Mascot
Base class: flash.display.MovieClip
Linkage: [✓ Export for ActionScript]
Export for runtime sharing
[✓ Export in first frame]
Import for runtime sharing
URL:

Figure 2.2
The Linkage Properties dialog is set to allow the `Mascot` *movie clip to be used by ActionScript.*

NOTE
I usually set the class name the same as the movie clip name. It makes it easier to remember.

Now we can create new copies of the movie clip using only ActionScript. The way this is done is to create a variable to hold the instance of the object, and then use `addChild` to put it in a display list:

```
var myMovieClip:Mascot = new Mascot();
addChild(myMovieClip);
```

Because we haven't set any other properties of the movie clip, it will appear at location 0,0 on the stage. We can set its location using the `x` and `y` properties of the instance. We can also set its angle of rotation using the `rotation` property. The value is in degrees:

```
var myMovieClip:Mascot = new Mascot();
myMovieClip.x = 275;
myMovieClip.y = 150;
myMovieClip.rotation = 10;
addChild(myMovieClip);
```

Although this looks like a lot of work for only one movie clip, ActionScript makes it easy to add multiple copies of a movie clip. The following code will create 10 copies of the `Mascot`, with horizontal locations changing from left to right by 50 pixels. It will also set the scale of the movie clips to 50 percent:

```
for(var i=0;i<10;i++) {
     var mascot:Mascot = new Mascot();
     mascot.x = 50*i+50;
     mascot.y = 300;
     mascot.scaleX = .5;
     mascot.scaleY = .5;
     addChild(mascot);
}
```

You can see the result of both pieces of code in Figure 2.3. The first `Mascot` is at the top, at location 275,100. The other `Mascots` are spread out from 50 to 500 at vertical location 300, and scaled 50 percent.

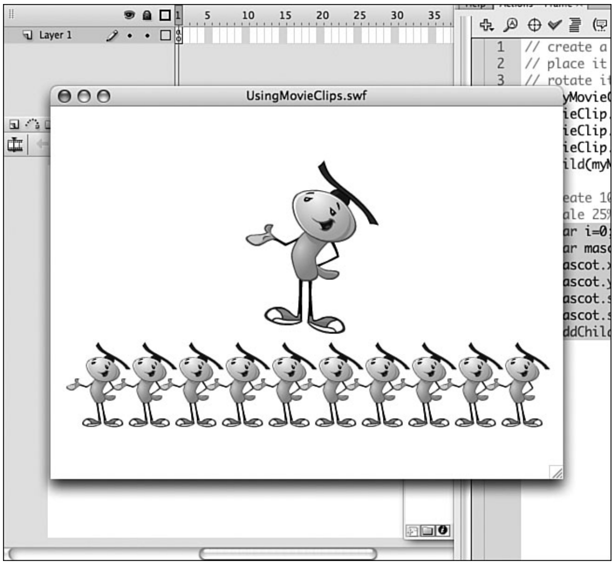


Figure 2.3
Eleven mascots are created and placed by ActionScript code.

You can find this example in the movie **`UsingMovieClips.fla`**. The code is in frame 1.

Making Buttons

You can also create buttons using only ActionScript. They can be made from either movie clips or button symbols stored in the library.

To make a movie clip into a clickable button, you only need to assign it a *listener.* This will allow the movie clip to accept events, in this case a mouse click event:

The following code places a new movie clip at 100,150:

```
var myMovieClip:Mascot = new Mascot();
myMovieClip.x = 100;
myMovieClip.y = 150;
addChild(myMovieClip);
```

To assign a listener, you use the `addEventListener` function. Include the type of event the listener should respond to. These are constant values that vary depending on the type of object and event. In this case, `MouseEvent.CLICK` will respond to a mouse click. Then, also include a reference to the function that you will create to handle the event (in this case, `clickMascot`):

```
myMovieClip.addEventListener(MouseEvent.CLICK, clickMascot);
```

The `clickMascot` function just sends a message to the Output window. Of course, in an application or game, it would do something more productive:

```
function clickMascot(event:MouseEvent) {
    trace("You clicked the mascot!");
}
```

One more thing you might want to do to make the movie clip more button-like is to set the `buttonMode` property of the movie clip instance to `true`. This will make the cursor switch to a finger cursor when the user rolls over it:

```
myMovieClip.buttonMode = true;
```

Of course, you can also create instances of button symbols using ActionScript. We do this in the same way as we did with movie clips. In this case, the symbol is linked as the class `LibraryButton`:

```
var myButton:LibraryButton = new LibraryButton();
myButton.x = 450;
myButton.y = 100;
addChild(myButton);
```

The main difference between movie clips and button symbols is that the buttons have four specialized frames in their timeline. Figure 2.4 shows the timeline of our `LibraryButton` symbol.

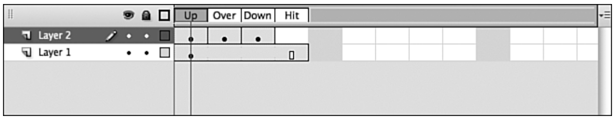


Figure 2.4
The timeline for a button contains four frames representing the three button states and a hit area.

The first frame represents the appearance of the button when the cursor is not over it. The second frame is what the button looks like when the cursor is hovering over it. The third frame is what the button looks like when the user has pressed down on the button, but has not yet released the mouse button. The last frame is the clickable area of the button. It is not visible at any time.

NOTE

The last frame can have a larger image than the rest to allow the user to click on or near the button. Or, if the visible frames of the button have gaps in them, such as they are just letters, or are an odd shape, the last frame can present a more standard circular or rectangular shape to represent the click area. You can also create invisible buttons by placing nothing on any of the frames except the last one.

Figure 2.5 shows the three button states and the hit area for a movie clip. This is just one example. Your button can show over and down states in any number of ways.

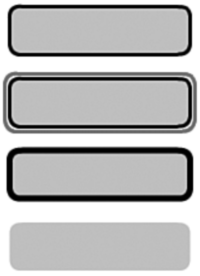


Figure 2.5
The four frames that make up a button symbol.

You can add a listener to the button in exactly the same way as you can with the movie clip:

```
myButton.addEventListener(MouseEvent.CLICK, clickLibraryButton);
function clickLibraryButton(event:MouseEvent) {
    trace("You clicked the Library button!");
}
```

The third option for creating a button is to use the `SimpleButton` type to create a button from scratch. Well, not exactly from scratch. You need to have a movie clip for each of the four frames of the button: Up, Over, Down, and Hit. So, in fact, you need four library elements, instead of just one.

To make this type of button, you use the `SimpleButton` constructor. Each of the four parameters for `SimpleButton` must be a movie clip instance. In this case, we will use four movie clips: `ButtonUp`, `ButtonOver`, `ButtonDown`, and `ButtonHit`:

```
var mySimpleButton:SimpleButton = new SimpleButton(new ButtonUp(),
        new ButtonOver(), new ButtonDown(), new ButtonHit());
mySimpleButton.x = 450;
mySimpleButton.y = 250;
addChild(mySimpleButton);
```

NOTE

You could also use the same movie clip for more than one of the four parameters in `SimpleButton`. For instance, you could reuse the button up state movie clip for the button hit movie clip. In fact, you could use the same movie clip for all four. This would make a less interesting button, but one that required fewer movie clips in the library.

Once again, you can add a listener to the button you created with the `addEventListener` command:

```
mySimpleButton.addEventListener(MouseEvent.CLICK, clickSimpleButton);
function clickSimpleButton(event:MouseEvent) {
     trace("You clicked the simple button!");
}
```

The movie **`MakingButtons.fla`** includes the code for all three of these buttons and will send a different message to the Output panel when each one is pressed.

Drawing Shapes

Not all the elements on the screen need to come from the library. You can use ActionScript 3.0 to draw with lines and basic shapes.

Every display object has a graphics layer. You can access it with the `graphics` property. This includes the stage itself, which you can access directly when writing code in the main timeline.

To draw a simple line, all you need to do is first set the line style, move to the starting point for the line, and then draw to an end point:

```
this.graphics.lineStyle(2,0x000000);
this.graphics.moveTo(100,200);
this.graphics.lineTo(150,250);
```

This sets the line style to 2 pixels thick and the color black. Then, a line is drawn from 100,200 to 150,250.

NOTE

Using the `this` keyword isn't necessary. But, when you want the line to be drawn in a specific movie clip instance, you need to specify it by name. For instance:

```
myMovieClipInstance.graphics.lineTo(150,250);
```

So, we'll include the `this` here to remind us of that, and make the code more reusable in your projects.

You can also create a curved line with `curveTo`. We'll have to specify both an end point and an anchor point. This gets rather tricky if you are not familiar with how Bezier curves are created. I had to guess a few times to figure out that this is what I wanted:

```
this.graphics.curveTo(200,300,250,250);
```

And then, we'll complete the line sequence with another straight line:

```
this.graphics.lineTo(300,200);
```

Now we have the line shown in Figure 2.6, which shows a straight line, then a curve, and then back up to a straight line.

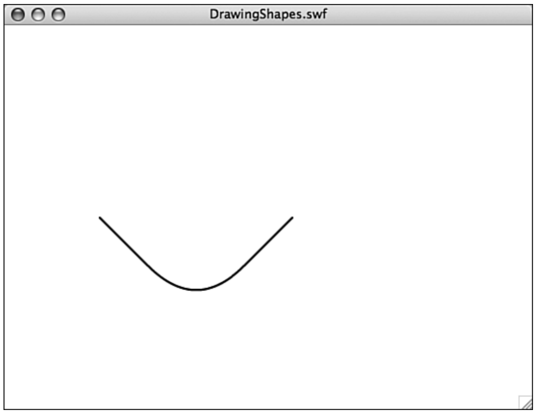


Figure 2.6
A line, curve, and a line make up this drawing.

You can also draw shapes. The simplest is a rectangle. The `drawRect` function takes a position for the upper-left corner, and then a width and a height:

```
this.graphics.drawRect(50,50,300,250);
```

You can also draw a rectangle with rounded edges. The extra two parameters are the width and height of the curved corners:

```
this.graphics.drawRoundRect(40,40,320,270,25,25);
```

A circle and an ellipse are also possible. The `drawCircle` takes a center point and a radius as the parameters:

```
this.graphics.drawCircle(150,100,20);
```

However, the `drawEllipse` function takes the same upper left and size parameters as `drawRect`:

```
this.graphics.drawEllipse(180,150,40,70);
```

You can also create filled shapes by starting with a `beginFill` function and the color of the fill:

```
this.graphics.beginFill(0x333333);
this.graphics.drawCircle(250,100,20);
```

To stop using a fill, you issue an `endFill` command.

Figure 2.7 shows the results of all of the drawing we have done.

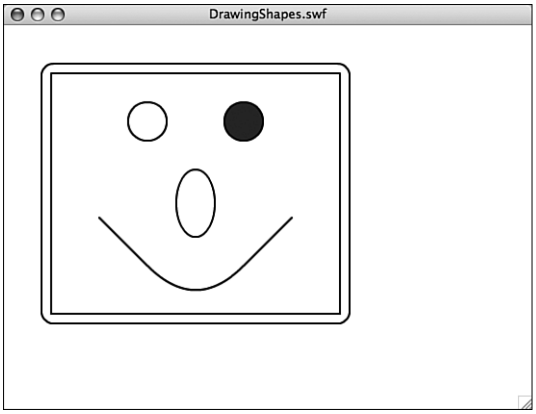


Figure 2.7
Two lines, a curve, a circle, ellipse, filled circle, rectangle, and rounded rectangle.

Most of these drawing functions have more parameters. For instance, `lineStyle` can also take an alpha parameter to draw a semitransparent line. Check the documentation for each of these functions if you want to know more.

The preceding examples can be found in **`DrawingShapes.fla`**.

Drawing Text

The Hello World examples in Chapter 1, "Using Flash and ActionScript 3.0," showed how you could create `TextField` objects to put text on the screen. The process involves creating a new `TextField` object, setting its `text` property, and then using `addChild` to add it to the stage:

```
var myText:TextField = new TextField();
myText.text = "Check it out!";
addChild(myText);
```

You can also set the location of the field with the `x` and `y` properties:

```
myText.x = 50;
myText.y = 50;
```

Likewise, you can set the width and height of the field:

```
myText.width = 200;
myText.height = 30;
```

It can be difficult to see the boundaries of a text field. A width of 200 might seem like enough to hold the current text, but will it hold different text if you change it? A quick way to see the actual size of a text field is to set the `border` property to `true` while you are testing:

```
myText.border = true;
```

Figure 2.8 shows the text field with the `border` turned on so that you can see its size.

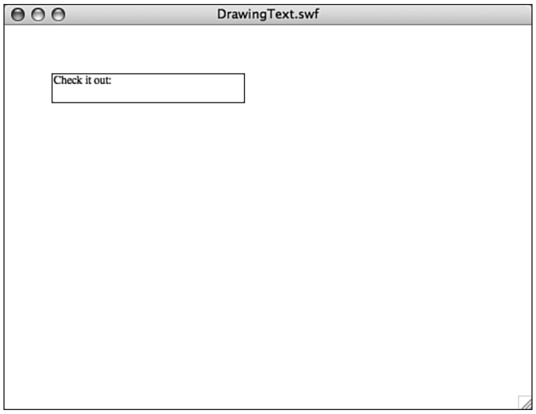


Figure 2.8
A text field at 50,50 with a width of 200 and a height of 30.

Another property we should almost always address is `selectable`. In most cases, you won't want this turned on, although it is the default. Leaving this property on means that the player's cursor will turn to a text editing cursor when he hovers over the text giving him the ability to select it:

```
myText.selectable = false;
```

What you most likely want to do when creating text is explicitly set the font, size, and style of the text. We can't do this directly. Instead, we need to create a `TextFormat` object. Then, we can set its `font`, `size` and `bold` properties:

```
var myFormat:TextFormat = new TextFormat();
myFormat.font = "Arial";
myFormat.size = 24;
myFormat.bold = true;
```

NOTE

You can also create a `TextFormat` object with just one line of code. For instance, the previous example could be done with this:

```
var myFormat:TextFormat = new TextFormat("Arial", 24, 0x000000, true);
```

The `TextFormat` constructor function accepts up to 13 parameters, but `null` can be used to skip any of them. Consult the documentation for a complete list.

Now that we have a `TextFormat` object, there are two ways to use it. The first is to use `setTextFormat` on a `TextField`. This changes the text to use the current style. However, you'd need to apply it each time you change the `text` property of the field.

A better way is to use `defaultTextStyle`. You do this before you set the `text` property. The next text will take on the style properties described in the `TextFormat`. In fact, every time you set the `text` of that `TextField`, you'll be using the same style. This is what we want most of the time in game development use of text fields:

```
myText.defaultTextFormat = myFormat;
```

Figure 2.9 shows the field with the format set.

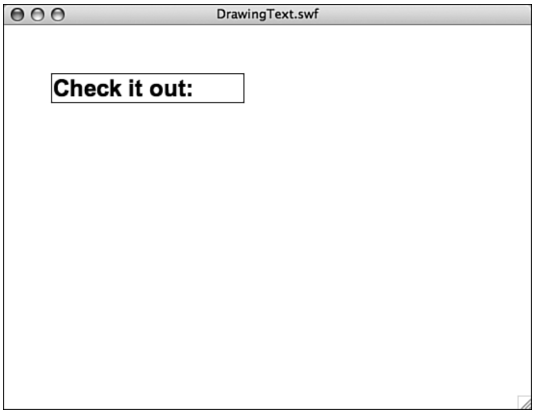


Figure 2.9
The text format has been set to 24-point Arial and bold.

You will want to check out many other properties of `TextFormat` in the documentation if you plan on stretching its abilities. You can also choose to use `StyleSheet` objects and HTML marked-up text set through the `htmlText` property of a `TextField`. The functionality with style sheets is very deep, so check the documentation if you want to investigate further.

Creating Linked Text

What do you get when you cross a text field and a button? A hypertext link, of course. You can easily make these in ActionScript 3.0, too.

The easiest way to create linked text in a `TextField` is by using the `htmlText` property of the field and passing in HTML code rather than the plain text used by the `text` property:

```
var myWebLink:TextField = new TextField();
myWebLink.htmlText = "Visit <A HREF='http://flashgameu.com'>FlashGameU.com</A>!";
addChild(myWebLink);
```

This works just like it would in a web page, except there is no default style change for the link. It appears in the same color and style as the rest of the text. But, when the user clicks the link, it will navigate away from the current page in the user's web browser to the page specified in the link.

NOTE

If the Flash movie is running as a stand-alone Flash projector, clicking this link would launch the user's browser and take the user to the web page specified. You can also specify the `TARGET` parameter of the `A` tag, if you are familiar with it in your HTML work. You can use `_top`, for instance, to specify the entire page, as opposed to the frame, or `_blank` to open up a blank window in the browser.

If you would like the text to appear blue and underlined, as it might on a web page, you can set a quick style sheet up and set the `styleSheet` property before setting the `htmlText`:

```
var myStyleSheet:StyleSheet = new StyleSheet();
myStyleSheet.setStyle("A",{textDecoration: "underline", color: "#0000FF"});
var myWebLink:TextField = new TextField();
myWebLink.styleSheet = myStyleSheet;
myWebLink.htmlText = "Visit <A HREF='http://flashgameu.com'>FlashGameU.com</A>!";
addChild(myWebLink);
```

Figure 2.10 shows the text using both the `textFormat` property set to Arial, 24, bold, and the `styleSheet` set to turn the links blue and underlined.

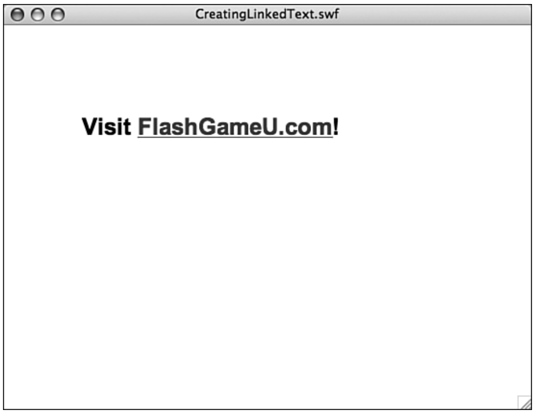


Figure 2.10
Both `defaultTextFormat` *and* `styleSheet` *have been used to format the text and the link.*

But, you don't need to have your links go to web pages. You can use them just like buttons, assigning listeners to the text fields that react to them.

To do this, you just need to use `event:` in the HREF tag of the link. Then, supply some text for your listener function to receive:

```
myLink.htmlText = "Click <A HREF='event:testing'>here</A>";
```

The listener will get the "testing" text as a string in the `text` property of the event that is returned:

```
addEventListener(TextEvent.LINK, textLinkClick);
function textLinkClick(event:TextEvent) {
    trace(event.text);
}
```

So, you could set several links in a `TextField` and then sort out which link has been clicked using the `text` property of the event parameter. Then, you could basically use text links just like you use buttons.

You can also style the text with `defaultTextFormat` and `styleSheet` just like the web link. The file **`CreatingLinkedText.fla`** includes examples of both types of links using the same format and style.

Creating Sprite Groups

Now that we know how to create a variety of screen elements, we can look a little deeper into how display objects and display lists work. We can create `Sprite` display objects that have no purpose other than to hold other display objects.

The following code will create a new `Sprite` and draw a 200x200 rectangle in it. The rectangle will have a 2-pixel black border and a light gray fill:

```
var sprite1:Sprite = new Sprite();
sprite1.graphics.lineStyle(2,0x000000);
sprite1.graphics.beginFill(0xCCCCCC);
sprite1.graphics.drawRect(0,0,200,200);
addChild(sprite1);
```

We can then position the `Sprite`, including the shape we drew inside it, to 50,50 on the stage:

```
sprite1.x = 50;
sprite1.y = 50;
```

Now we'll create a second `Sprite`, just like the first, but position it at 300,100:

```
var sprite2:Sprite = new Sprite();
sprite2.graphics.lineStyle(2,0x000000);
sprite2.graphics.beginFill(0xCCCCCC);
sprite2.graphics.drawRect(0,0,200,200);
sprite2.x = 300;
sprite2.y = 50;
addChild(sprite2);
```

Now let's create a third `Sprite`, this time containing a circle. Instead of using `addChild` to place it on the stage, however, we'll place it inside `sprite1`. We'll also give it a darker fill:

```
var sprite3:Sprite = new Sprite();
sprite3.graphics.lineStyle(2,0x000000);
sprite3.graphics.beginFill(0x333333);
sprite3.graphics.drawCircle(0,0,25);
sprite3.x = 100;
sprite3.y = 100;
sprite1.addChild(sprite3);
```

Figure 2.11 shows what the three sprites look like on the screen. Notice that even though we set the `x` and `y` properties of the circle to 100,100, it does not appear at 100,100 relative to the stage, but rather it is at 100,100 inside of `sprite1`.

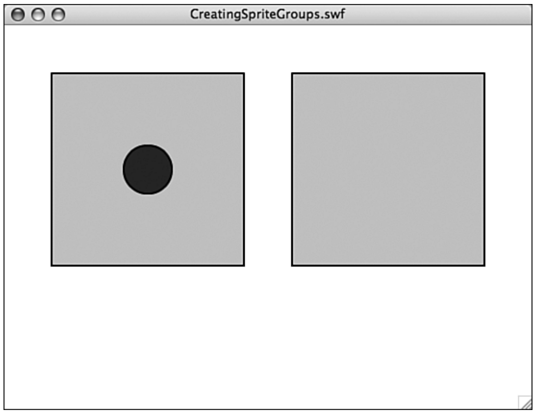


Figure 2.11
The circle sprite is inside the left rectangle's sprite.

The movie now has `sprite1` and `sprite2` as children of the stage. Then, `sprite3` is a child of `sprite1`. If we make `sprite3` a child of `sprite2` instead, it will jump to the center of `sprite2` because the 100,100 position of `sprite3` is now relative to its new parent.

The movie **`CreatingSpriteGroups.fla`** makes it easier to visualize this by placing a listener on both `sprite1` and `sprite2`. When you click either of them, `sprite3` is set to its child. So, you can make `sprite3` jump back and forth between parents:

```
sprite1.addEventListener(MouseEvent.CLICK, clickSprite);
sprite2.addEventListener(MouseEvent.CLICK, clickSprite);
function clickSprite(event:MouseEvent) {
    event.currentTarget.addChild(sprite3);
}
```

NOTE

This is also a good example of how one button listener can be used for multiple buttons. The actual object clicked is passed into the listener function via `currentTarget`. In this case, we can just use that value for `addChild`. However, you can also compare it to a list of possible objects clicked and execute code based on which object was pressed.

In game development, we'll be creating `Sprite` groups all the time to hold different types of game elements. If we are using `Sprites` simply for layering, we'll keep them all at 0,0, and then we can move elements from `Sprite` to `Sprite` without changing their relative screen position.

Setting Sprite Depth

Worth mentioning at this point is the `setChildIndex` command. This allows you to move display objects up and down in the display list. In other words, you can put one `Sprite` on top of the other.

Think of the display list as an array, starting with item 0. If you have created three `Sprites`, they are at positions 0, 1, and 2. Position 2 is the top `Sprite` and is drawn on top of the others.

If you ever want to move a `Sprite` to the bottom, under all other `Sprites`, just use the following:

```
setChildIndex(myMovieClip,0);
```

This puts the `myMovieClip` display object at position 0, and then all the rest move up to fill in the gap it left behind.

Setting a `Sprite` to be higher than all others is a little trickier. You need to set the index to the last item in the display list. So, if there are three items (0, 1, and 2), you need to set it to 2. This can be done with the `numChildren` property:

```
setChildIndex(myMovieClip,numChildren-1);
```

You need to use the -1 because if there are three children (0, 1, and 2), `numChildren` returns 3. However, we need to use 2 in `setChildIndex`. Using 3 will give us an error.

The example movie **`SettingSpriteDepth.fla`** puts three `Sprites` on the screen, each overlapping the other. Then, you can click any one to set it to the top position.

Accepting Player Input

The following sections deal with getting input from the player. This always comes from the keyboard or the mouse because these are the only standard input devices on modern computers.

Mouse Input

We already know quite well how to turn a `Sprite` into a button and have it react to mouse clicks. But, the mouse is good for more than just clicking. You can also get the cursor's location at any time, and `Sprites` can detect whether the cursor is over them.

You can access the cursor's current stage location at any time with the `mouseX` and `mouseY` properties. The following code takes the current location of the cursor and places it in a text field every frame:

```
addEventListener(Event.ENTER_FRAME, showMouseLoc);
function showMouseLoc(event:Event) {
    mouseLocText.text = "X="+mouseX+" Y="+mouseY;
}
```

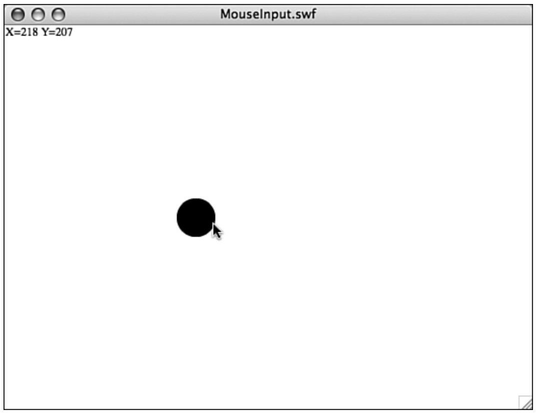
You can detect when the cursor moves over a `Sprite` in a similar manner to how you can detect a mouse click. Instead of a click, we are looking for a rollover event. We can add a listener for it to the `Sprite`:

```
mySprite.addEventListener(MouseEvent.ROLL_OVER, rolloverSprite);
function rolloverSprite(event:MouseEvent) {
    mySprite.alpha = 1;
}
```

In this function, we set the `alpha` property of the `Sprite` to 1, which is 100 percent opaque. Then, when the cursor leaves the `Sprite`, we'll set it to 50 percent:

```
mySprite.addEventListener(MouseEvent.ROLL_OUT, rolloutSprite);
function rolloutSprite(event:MouseEvent) {
    mySprite.alpha = .5;
}
```

In the movie **`MouseInput.fla`**, the Sprite starts off at 50 percent transparency, and only changes to 100 percent when the cursor is over the `Sprite`. Figure 2.12 shows both the text field read-out of the cursor location, and this `Sprite`.



```
X=218 Y=207
```

Figure 2.12
The cursor is over the sprite, so it turns opaque.

Keyboard Input

Detecting keyboard input relies on the two keyboard events: `KEY_UP` and `KEY_DOWN`. When the user presses down on a key, the `KEY_DOWN` message is sent. If you set a listener to listen for it, you'll be able to do something with it.

However, the addEventListener function must reference the stage object. This is because key presses don't have an obvious target like mouse clicks do. So, there must be an object that has keyboard focus when a movie starts. The stage is that object:

```
stage.addEventListener(KeyboardEvent.KEY_DOWN, keyDownFunction);
```

When a function gets the call from this listener, it will be able to access many properties of the event parameter. One such parameter is charCode, which returns the character number of the key pressed.

In the following example, the charCode is converted to a character, and then is displayed in the text field keyboardText:

```
function keyDownFunction(event:KeyboardEvent) {
     keyboardText.text = "Key Pressed: "+String.fromCharCode(event.charCode);
}
```

NOTE

Remember to choose Control, Disable Keyboard Shortcuts from the menu while testing. Otherwise, your key presses might not get through to the stage at all.

The event properties also include keyCode, which is like charCode, but is not affected by the Shift key. For instance, with the Shift key pressed, the A key will give us a charCode of 65 for a capital A. But, with the Shift key released, it gives us a charCode of 97, representing a lowercase A. The keyCode returns 65 regardless.

Other properties of the event include ctrlKey, shiftKey, and altKey, representing whether those modified keys are depressed.

In games, we often don't care about the initial key press, but about whether the player continues to hold down a key. For instance, in a driving game, we want to know whether the player has the accelerator pedal pressed down, represented by the up arrow.

To recognize when a key is being held down, the strategy is to look for both KEY_DOWN and KEY_UP. When we detect that a key is pressed down, we set a Boolean variable to true. Then, when the same key is lifted up, we set it to false. To determine whether the key is pressed at any given time, all we need to do is check the Boolean variable.

Here is some code that checks the spacebar in the same way. The first function checks to see when the spacebar is pressed, and sets the variable spacePressed to true:

```
stage.addEventListener(KeyboardEvent.KEY_DOWN, keyDownFunction);
function keyDownFunction(event:KeyboardEvent) {
     if (event.charCode == 32) {
          spacebarPressed = true;
     }
}
```

This next function captures a key being lifted up. If this is the spacebar, `keyPressed` is set to `false`:

```
stage.addEventListener(KeyboardEvent.KEY_UP, keyUpFunction);
function keyUpFunction(event:KeyboardEvent) {
     if (event.charCode == 32) {
          spacebarPressed = false;
     }
}
```

Using this method, we can keep track of critical game keys, like the spacebar and the four arrow keys. The example movie **`KeyboardInput.fla`** keeps track of the spacebar in this way and also displays a message when the spacebar's state changes.

Text Input

Another type of `TextField` object is an input field. The difference between either a static or dynamic text field and the input field is that the user can select and type into the input field. To create a `TextField` that is an input field, just set its `type` property:

```
var myInput:TextField = new TextField();
myInput.type = TextFieldType.INPUT;
addChild(myInput);
```

This creates a hard-to-find and poorly shaped input field at the upper-left corner of the screen. We can do a lot to improve it by setting its properties and using a `TextFormat` object.

The following code sets the format to 12-point Arial, positions the field to 10,10 with a height of 18 and a width of 200. It also turns on the `border` as you would expect with an input field in a typical piece of software:

```
var inputFormat:TextFormat = new TextFormat();
inputFormat.font = "Arial";
inputFormat.size = 12;

var myInput:TextField = new TextField();
myInput.type = TextFieldType.INPUT;
myInput.defaultTextFormat = inputFormat;
myInput.x = 10;
myInput.y = 10;
myInput.height = 18;
myInput.width = 200;
myInput.border = true;
addChild(myInput);
stage.focus = myInput;
```

The last line of code places the text entry cursor into the field.

A typical `TextField` is set to be a single line of text. You can change this behavior using the `multiline` property. For most text input, however, we'll want only a single line. This means that Return/Enter keys will not be recognized, because a second line of text can't be created. However, we can capture this key and use it to signal the end of input.

To capture the Return key, we place a listener on the key up event. Then, the responding function will check to see whether the key pressed was code number 13, the Return key:

```
myInput.addEventListener(KeyboardEvent.KEY_UP, checkForReturn);
function checkForReturn(event:KeyboardEvent) {
     if (event.charCode == 13) {
          acceptInput();
     }
}
```

The `acceptInput` function takes the text from the input field and stores it in `theInputText`. Then it sends it to the Output window. It also removes the text field:

```
function acceptInput() {
     var theInputText:String = myInput.text;
     trace(theInputText);
     removeChild(myInput);
}
```

NOTE

While testing this movie, I found that the test environment sometimes intercepted the Return key, even if I had the Disable Keyboard Shortcuts menu option turned on. Clicking in the window and trying again produced the desired result. This shouldn't happen when the movie is deployed on the Web.

The sample movie **`TextInput.fla`** contains the preceding code in action.

Creating Animation

Next, let's take a look at some ActionScript that enables us to move `Sprites` around the screen. And, let's look at methods for making this movement mimic the real world.

Sprite Movement

Changing the position of a `Sprite` or movie clip is as easy as setting its `x` or `y` position. To animate one, therefore, we just need to change it at a constant rate.

Using the `ENTER_FRAME` event, we can easily program this kind of constant change. For instance, here is a short program that makes a copy of a movie clip in the library, and then moves it one pixel to the right every frame:

```
var hero:Hero = new Hero();
hero.x = 50;
hero.y = 100;
addChild(hero);

addEventListener(Event.ENTER_FRAME, animateHero);
function animateHero(event:Event) {
    hero.x++;
}
```

The hero character now slides across the stage, one pixel at a time. Instead of moving one pixel at a time, you could move, say, ten at a time with `+= 10` rather than `++`.

Another way to speed up the hero is to simply increase the frame rate. Instead of the default 12fps (frames per second), you can increase it to, say, 60fps. You do this in the Properties Inspector at the upper-right corner, if the stage is selected, not another movie clip. Figure 2.13 shows the Property Inspector with the Frame rate set to 12fps.

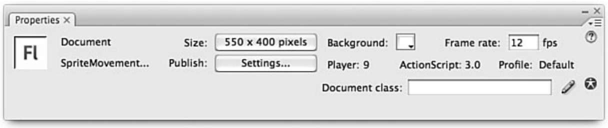

Properties ×
Fl Document SpriteMovement...
Size: 550 x 400 pixels
Publish: Settings...
Background:
Frame rate: 12 fps
Player: 9 ActionScript: 3.0 Profile: Default
Document class:

Figure 2.13
The Property Inspector enables you to change the frame rate of the movie.

NOTE
Just because you choose to have a frame rate of 60fps, doesn't mean the movie runs at 60fps. It just tries. If there is a lot going on in the movie, and the user's machine is slow, it will fall short of 60fps. We'll be looking at time-based animation soon, which will offer a better alternative to frame-based animation.

Instead of just having the hero slide across the screen, we can make him walk. We need the help of an animation artist, however. He or she will need to create several frames of a walk cycle and place them in sequential frames of the `Hero` movie clip. Figure 2.14 shows such a walk cycle.

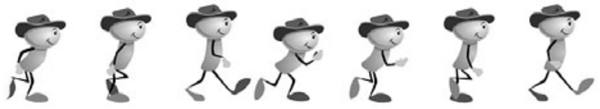


Figure 2.14
A simple looping walk cycle in seven frames.

Now we just have to rewrite the animateHero function so that the character moves from frame 2 to frame 8 of the movie clip, which represent the seven frames of animation. Frame 1 is reserved as the "stand" position.

We'll also be told by the artist that the character should move about 7 pixels per frame horizontally to match the walking animation.

The resulting code will check the currentFrame property of the movie clip, and if it is on frame 8, it will send the movie clip back to frame 2. Otherwise, it will just go to the very next frame:

```
function animateHero(event:Event) {
     hero.x += 7;
     if (hero.currentFrame == 8) {
     hero.gotoAndStop(2);
     } else {
     hero.gotoAndStop(hero.currentFrame+1);
     }
}
```

Try the example movie **SpriteMovement.fla** to see this code in action. Try different movie frame rates to see him move faster or slower.

Using Timers

Think of Timers like little message clocks. You create one and start it, and then it ticks away and sends messages at a set interval. For instance, you could create a Timer to call a specific function every second.

The way you set up a Timer is to create a new Timer object. You need to pass to it the number of milliseconds between events. You can also pass a second parameter for the number of events to generate before stopping, although we won't use that here.

The following code creates a new Timer that triggers an event every 1,000 milliseconds (every 1 second). It calls the function timerFunction for each of these events:

```
var myTimer:Timer = new Timer(1000);
myTimer.addEventListener(TimerEvent.TIMER, timerFunction);
```

To test the Timer, we just have it draw a small circle with every event. The event parameter sent into the function includes a target property that refers to the Timer. You can use this to get at the currentCount property, which is the number of times the Timer has been triggered. We'll use this to offset each circle so that it draws a line of circles, from left to right:

```
function timerFunction(event:TimerEvent) {
     this.graphics.beginFill(0x000000);
     this.graphics.drawCircle(event.target.currentCount*10,100,4);
}
```

Just creating the `Timer` and attaching a listener to it is not enough. You also have to tell the `Timer` to begin. You can do that with the `start()` command:

```
myTimer.start();
```

The movie **`UsingTimers.fla`** demonstrates the preceding code.

You can also use a `Timer` to accomplish the same tasks as we did in the previous section with the `enterFrame` events. Here is a `Timer` that will call the same `animateHero` function to move the character across the screen with a walk cycle. It replaces the `addEventListener` call:

```
var heroTimer:Timer = new Timer(80);
heroTimer.addEventListener(TimerEvent.TIMER, animateHero);
heroTimer.start();
```

You can see this code in action in **`UsingTimers2.fla`**. When you run it, the character walks at a similar pace to 12fps. But, you can set the movie's frame rate to 12, 6, or 60 and the walk will move at the same rate.

NOTE

Try setting the frame rate to 1fps. With the `Timer` moving the character every 80 milliseconds, it will move the character quite a bit between screen updates. This shows, at least, that `Timers` can be used to provide consistent movement despite a slower machine, so long as the calculations being performed with each `Timer` event are not overtaxing the processor.

Time-Based Animation

Real time-based animation means animation steps based on how much time has passed, not on arbitrary time intervals.

A time-based animation step would first calculate the time since the last step. Then, it would move the objects according to this time difference. For instance, if the first time interval is .1 seconds, and the second time interval is .2 seconds, objects will move twice as far after the second time interval to remain consistent.

So, the first thing that must be done is a variable should be created that holds the time of the last step. We'll start it by placing the current time, taken from the `getTimer()` system function. This returns the time since the Flash player started, in milliseconds:

```
var lastTime:int = getTimer();
```

Then, we'll create an event listener tied to the `ENTER FRAME` event. It calls `animateBall`:

```
addEventListener(Event.ENTER_FRAME, animateBall);
```

The `animateBall` function calculates the time difference, and then sets the `lastTime` variable to get ready for the next step. It then sets the `x` location of a movie clip instance `ball`. It adds the `timeDiff` multiplied by .1. So, the movie clip will move 100 pixels every 1000 milliseconds:

```
function animateBall(event:Event) {
     var timeDiff:int = getTimer()-lastTime;
     lastTime += timeDiff;
     ball.x += timeDiff*.1;
}
```

The movie **`TimeBasedAnimation.fla`** uses this code to move a ball across the screen. Try it first at 12fps. Then try it at 60fps. Notice how the ball gets to the other side of the screen at the same time, but it looks a lot better at 60fps.

Physics-Based Animation

With ActionScript animation, you can do more than make an object move along a pre-determined path. You can also give it physical properties and have it move like an object in the real world.

Physics-based animation can be either frame based or time based. We'll build on the time-based animation example, but using velocity and gravity to dictate where the object should move.

NOTE

Gravity is a constant acceleration toward the ground (in this case, the bottom of the screen). In the real world, gravity is 9.8 meters/second or 32 feet/second. In the Flash Player world, everything is measured in pixels per millisecond, which can be scaled to match the real world in any way you want. For instance, if 1 pixel is 1 meter, .0098 is .0098 meters/millisecond, or 9.8 meters/second. However, you could just as easily used .001 or 7 or any number, as long as it "looks" right in your game. We're not building scientific simulations here, just games.

We'll set the gravity to .0098 and define a starting velocity for the moving element. Velocity is simply the speed and direction of a moving object. With `dx` and `dy` representing the change in the horizontal and change in the vertical position, together they represent velocity:

```
// set gravity amount
var gravity:Number = .00098;
var b:Number = .05;
// set starting velcity
var dx:Number = .2;
var dy:Number = -.8;
```

So, the object, in this case a ball, should move .2 pixels horizontally every millisecond, and –.8 pixels vertically every millisecond. Which means it is being thrown up and somewhat to the right.

To control the animation, we'll create an `ENTER FRAME` listener and initialize the `lastTime` variable:

```
// mark start time and add listener
var lastTime:int = getTimer();
addEventListener(Event.ENTER_FRAME, animateBall);
```

The `animateBall` function starts by calculating the amount of time passed since the last animation step:

```
// step animation
function animateBall(event:Event) {

     // get time difference
     var timeDiff:int = getTimer()-lastTime;
     lastTime += timeDiff;
```

The `dy` variable is the vertical speed, and it should change depending on the pull of gravity, metered by the time difference:

```
     // adjust vertical speed for gravity
     dy += gravity*timeDiff;
```

The ball moves according to the two variables: `dx` and `dy`. In both cases, the `timeDiff` is used to determine how much:

```
     // move ball
     ball.x += timeDiff*dx;
     ball.y += timeDiff*dy;
}
```

When you run the movie **`PhysicsBasedAnimation.fla`**, you get results that look like the ones shown in Figure 2.15.

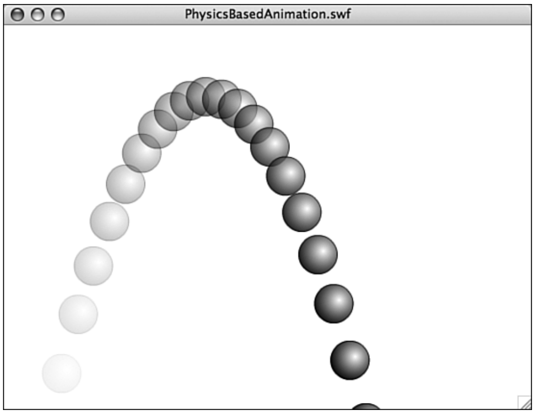


Figure 2.15
This time-lapse screen shot shows the positions of the ball at 12fps.

Programming User Interaction

Beyond the basic user input and sprite movement, we've got the combination of both—when user interaction affects the elements on the screen. The following programs are small examples of user interaction with sprites.

Moving Sprites

Sprites on the screen are typically moved with either the mouse or the keyboard. For the keyboard, the arrow keys are typically used to control the sprite.

Earlier in this chapter, you saw how to determine whether the spacebar was pressed. We can use the same basic idea here to determine whether the arrow keys are pressed. although the arrow keys have no visible character representation, they can be represented by the key codes 37, 38, 39, and 40. Figure 2.16 shows the four arrow keys and their key codes.

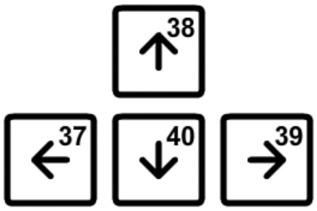


Figure 2.16
The four arrow keys can be referenced by these four key codes.

We start by creating four Boolean variables to hold the state of the four arrow keys:

```
// initialize arrow variables
var leftArrow:Boolean = false;
var rightArrow:Boolean = false;
var upArrow:Boolean = false;
var downArrow:Boolean = false;
```

We need both `KEY DOWN` and `KEY UP` listeners, as well as an `ENTER FRAME` listener to deal with moving the sprite as often as the screen updates:

```
// set event listeners
stage.addEventListener(KeyboardEvent.KEY_DOWN, keyPressedDown);
stage.addEventListener(KeyboardEvent.KEY_UP, keyPressedUp);
stage.addEventListener(Event.ENTER_FRAME, moveMascot);
```

When the user presses an arrow key down, we'll set its Boolean variable to `true`:

```
// set arrow variables to true
function keyPressedDown(event:KeyboardEvent) {
    if (event.keyCode == 37) {
        leftArrow = true;
    } else if (event.keyCode == 39) {
        rightArrow = true;
```

```
     } else if (event.keyCode == 38) {
          upArrow = true;
     } else if (event.keyCode == 40) {
          downArrow = true;
     }
}
```

Likewise, when the user releases the arrow keys, we'll set the Boolean variable to `false`:

```
// set arrow variables to false
function keyPressedUp(event:KeyboardEvent) {
     if (event.keyCode == 37) {
          leftArrow = false;
     } else if (event.keyCode == 39) {
          rightArrow = false;
     } else if (event.keyCode == 38) {
          upArrow = false;
     } else if (event.keyCode == 40) {
          downArrow = false;
     }
}
```

Now we can use these Booleans to move the `mascot` movie clip by a set amount in the proper directions. We'll store the movement amount in `speed`, instead of repeating it four times in the code:

```
// move every frame
function moveMascot(event:Event) {
     var speed:Number = 5;

     if (leftArrow) {
          mascot.x -= speed;
     }
     if (rightArrow) {
          mascot.x += speed;
     }
     if (upArrow) {
          mascot.y -= speed;
     }
     if (downArrow) {
          mascot.y += speed;
     }
}
```

The movie **`MovingSprites.fla`** shows this code in action. Notice that because we are testing each arrow key Boolean separately, you can do combinations. For instance, you can press the right and down arrows, and the mascot will move down and to the right at the same time. Hold both the left and right arrows and the mascot won't move at all (because they cancel each other out).

Dragging Sprites

Another way to move a `Sprite` around the stage is to let the user click and drag it.

Instead of looking at the keyboard, we'll be looking at the mouse here. When the user clicks the sprite, we'll start the drag. When the user releases the mouse button, we'll stop the drag.

However, we can't rely on the cursor being over the sprite when the user releases the button. So, we'll look for the `MOUSE_DOWN` event on the `mascot` sprite, but the `MOUSE_UP` event on the `stage`. The `stage` will get a `MOUSE_UP` event regardless of whether the cursor is over the sprite:

```
// set listeners
mascot.addEventListener(MouseEvent.MOUSE_DOWN, startMascotDrag);
stage.addEventListener(MouseEvent.MOUSE_UP, stopMascotDrag);
mascot.addEventListener(Event.ENTER_FRAME, dragMascot);
```

Another factor is the cursor offset. We want to allow the user to drag the sprite from any point on the sprite. So, if the player clicks at the bottom-right corner of the sprite, the cursor and the bottom-right corner will continue to stay in the same position relative to each other as the user drags.

To do this, we'll figure out the offset between the 0,0 location of the sprite and the location of the mouse click and store it in `clickOffset`. We'll also use this variable to determine whether a drag is happening at the moment. If it is, `clickOffset` will be set to a `Point` object. If not, it should be `null`:

```
// offset between sprite location and click
var clickOffset:Point = null;
```

When the user clicks the sprite, the click offset is taken from the `localX` and `localY` properties of the click event:

```
// user clicked
function startMascotDrag(event:MouseEvent) {
    clickOffset = new Point(event.localX, event.localY);
}
```

When the user releases the cursor, the `clickOffset` is set back to `null`:

```
// user released
function stopMascotDrag(event:MouseEvent) {
    clickOffset = null;
}
```

Then, every frame, if the `clickOffset` is not `null`, we'll set the position of the `mascot` to the current cursor location, minus the offset:

```
// run every frame
function dragMascot(event:Event) {
     if (clickOffset != null) { // must be dragging
          mascot.x = mouseX - clickOffset.x;
          mascot.y = mouseY - clickOffset.y;
     }
}
```

Check out **`DraggingSprites.fla`** to see this code at work. Try dragging the mascot from different points to see how the `clickOffset` handles it.

Collision Detection

After you have objects moving around the screen in game, it is very common to test them for collisions against each other.

ActionScript 3.0 contains two native collision-detection functions. The `hitTestPoint` function tests a point location to see whether it is inside a display object. The `hitTestObject` function tests two display objects against each other to see whether they overlap.

To examine these two functions, let's create a simple example that examines the cursor location and the location of a moving sprite every frame:

```
addEventListener(Event.ENTER_FRAME, checkCollision);
```

The `checkCollision` function starts by using `hitTestPoint` looking at the cursor location to see whether it hits the `crescent` movie clip on the stage. The first two parameters of the `hitTestPoint` function are the x and y location of the point. The third location is the type of boundary to use. The default value of `false`, which means only the bounding rectangle of the display object should be taken into account.

Unless the sprite is somewhat box shaped, this isn't good enough for most game use. Instead, by setting the third parameter to `true`, `hitTestPoint` will use the actual shape of the display object to determine collision.

We'll put different text into a message text field, depending on the results of `hitTestPoint`:

```
function checkCollision(event:Event) {

     // check the cursor location against the crescent
     if (crescent.hitTestPoint(mouseX, mouseY, true)) {
          messageText1.text = "hitTestPoint: YES";
     } else {
          messageText1.text = "hitTestPoint: NO";
     }
```

The `hitTestObject` function doesn't have a shape option. It only compares the two bounding boxes of two sprites. But, it can still be useful in some cases.

The following part of the code makes a `star` movie clip follow the cursor, and puts a different message in another text field depending on whether their bounding boxes intersect:

```
     // move star with mouse
     star.x = mouseX;
     star.y = mouseY;

     // test star versus crescent
     if (star.hitTestObject(crescent)) {
          messageText2.text = "hitTestObject: YES";
     } else {
          messageText2.text = "hitTestObject: NO";
     }
}
```

The example movie **`CollisionDetection.fla`** showcases this example. Figure 2.17 shows the cursor is inside the bounding box of the crescent; because we're testing `hitTestPoint` with the shape flag set to `true`, however, it won't register a collision unless the cursor actually over the crescent. The star and the crescent, meanwhile, are colliding as their bounding boxes intersect.

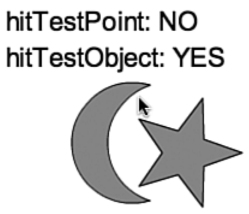

hitTestPoint: NO
hitTestObject: YES

Figure 2.17
The cursor location and the star are being tested for collision with the crescent.

Accessing External Data

Sometimes it is necessary to access information outside the game. You can load external game parameters from the web page or text fields. You can also save and load information locally.

External Variables

Suppose you have a game that could vary according to some options, such as a jigsaw puzzle game that could use different pictures, or an arcade game that could run at different speeds.

You can feed variable values into the Flash movie through the HTML page it is sitting on.

There are a few different ways to do this; but if you are using the default HTML template from the publish settings, you can pass in parameter values via the `flashvars` property of the `AC_FL_RunContent` function.

NOTE

The Flash movie is embedded into the web page via the `OBJECT` and `EMBED` tags for ActiveX (Internet Explorer) and Plug-In (Firefox) architectures. However, the `OBJECT` and `EMBED` tags are in turn written out by a bit of JavaScript provided by Adobe that comes in the **`AC_RunActiveContent.js`** file. Depending on your Publish Settings, you will get a copy of this file every time you Publish.

Here is a shortened version of the call to `AC_FL_RunContent` that you would find in the HTML file that publishing from Flash CS3 would export. It includes a `flashvars` parameter that I have added:

```
<script language="javascript">
AC_FL_RunContent(
     'codebase',
'http://download.macromedia.com/pub/shockwave/cabs/flash/swflash.cab#version=9,0,0,0',
     'width', '550',
     'height', '400',
     'src', 'ExternalVariables',
     'quality', 'high',
     'flashvars', 'puzzleFile=myfilename.jpg&difficultyLevel=7'
     );
</script>
```

The `flashvars` format is property name = value pairs, separated by the & symbol between them. So in this case, the property `puzzleFile` is set to `myfilename.jpg`, and the property `difficultyLevel` is set to `7`.

When the Flash movie starts, it can get these values using the `LoaderInfo` object. The following line retrieves all the parameters and places them in an object:

```
var paramObj:Object = LoaderInfo(this.root.loaderInfo).parameters;
```

To access an individual value, you just use ActionScript code like this:

```
var diffLevel:String = paramObj["difficultyLevel"];
```

You could pass in any number of game constants, such as image names, starting levels, speeds, positions, and so on. A hangman game could be set up with a different word or phrase. A world-exploring game could be given a different start location.

When running the movie **`ExternalVariables.fla`**, keep in mind that the right way to do it is to load the **`ExternalVariables.html`** page in your browser. This has the `flashvars` parameter all set. If you try to test in Flash, or try to create a new HTML page, those parameters will be missing.

Loading Data

Loading data from an external text file is relatively easy. If it is an XML-formatted file, it is ideal.

For instance, suppose you want to load a trivia question from a file. The XML data could look like this:

```
<LoadingData>
    <question>
        <text>This is a test</text>
        <answers>
            <answer type="correct">Correct answer</answer>
            <answer type="wrong">Incorrect answer</answer>
        </answers>
    </question>
</LoadingData>
```

To load the data, you use two objects: a `URLRequest` and a `URLLoader`. Then, you listen for the loading to be complete, and call a function of your own:

```
var xmlURL:URLRequest = new URLRequest("LoadingData.xml");
var xmlLoader:URLLoader = new URLLoader(xmlURL);
xmlLoader.addEventListener(Event.COMPLETE, xmlLoaded);
```

The `xmlLoaded`, in this case, is just some `trace` statements to show the data was coming in:

```
function xmlLoaded(event:Event) {
    var dataXML  = XML(event.target.data);
    trace(dataXML.question.text);
    trace(dataXML.question.answers.answer[0]);
    trace(dataXML.question.answers.answer[0].@type);
}
```

You can see how easy it is to get the XML data from the file. With the XML object being `dataXML`, you can retrieve the question text with `dataXML.question.text` and the first answer with `dataXML.question.answers[0]`. You can get an attribute, like the `type` of answer by using `@type`.

The **`LoadingData.fla`** example reads its data from the **`LoadingData.xml`** file. Try changing and adding to the data in the XML file. Then, try playing with the `trace` statements to access different parts of the data.

Saving Local Data

A common need in game development is to store bits of local data. For instance, you could store the player's previous score. Or, you could store some game options.

To store a piece of data on the user's machine, we'll use a local `SharedObject`. Accessing a `SharedObject` is the same act as creating one. Just asking whether it exists will create it.

To do this, just assign a variable to the `SharedObject` of a certain name, with the `getLocal` function:

```
var myLocalData:SharedObject = SharedObject.getLocal("mygamedata");
```

The `myLocalData` object is able to take any number of properties of any type: numbers, strings, arrays, other objects, and so on.

If you had stored same data in a property of the shared object named `gameinfo` you could access it with: `myLocalData.data.gameinfo`:

```
trace("Found Data: "+myLocalData.data.gameinfo);
```

So, set this `gameinfo` property. Just set it as you would a regular variable:

```
myLocalData.data.gameinfo = "Store this.";
```

Try running the test movie **`SavingLocalData.fla`**. It uses the `trace` function to output the `myLocalData.data.gameinfo` property. Because that isn't set to anything, you'll get `undefined` as the result. But then, it sets the value. So, the second time you run the test, you get "Store this."

Miscellaneous Game Elements

Here are some simple scripts that perform a variety of tasks. Most of these can be added to any game movie, if needed.

Custom Cursors

Suppose you want to replace the standard mouse cursor with something that fits the style of your game. Or, perhaps you want a larger cursor for a child's game, or a special cross-hair cursor for a shooting game.

Although you can't change the computer's cursor, you can make it disappear, at least visually. Then, you can replace it with a sprite that matches the cursor location and floats above everything else.

To make the cursor invisible, use the `Mouse.hide()` command:

```
Mouse.hide();
```

Then, to have a sprite act as the cursor, place it in a layer above all the other screen elements. Figure 2.18 shows the timeline with three layers. The cursor is the only element on the second layer, and all other elements are below it in the third layer.

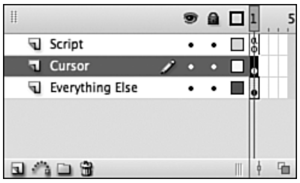


Figure 2.18
The cursor must remain on top of all other screen elements.

NOTE

If you are creating objects with ActionScript, you have to work to keep the cursor above all else. Using the `setChildIndex` command will allow you to place the cursor at the top after you have created your game objects.

To make a sprite follow the cursor, we need an `ENTER FRAME` listener:

```
addEventListener(Event.ENTER_FRAME, moveCursor);
```

Then, the `moveCursor` command will simply have the `arrow` object, which is the stage instance name of the cursor in this case, follow the mouse location:

```
function moveCursor(event:Event) {
     arrow.x = mouseX;
     arrow.y = mouseY;
}
```

You also need to set the `mouseEnabled` property of the sprite to `false`. Otherwise, the hidden cursor is always over the cursor sprite, and never over sprites under it, like a button:

```
arrow.mouseEnabled = false;
```

Without this line of code, you can move the mouse over a button, and the button will not show its rollover state or accept mouse clicks properly. But, this makes the custom cursor invisible to mouse events.

Figure 2.19 shows the custom cursor moving over a simple button.

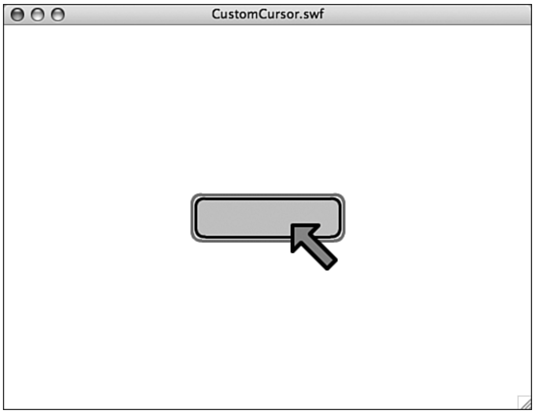


Figure 2.19
The simple button shows its rollover state, even though the arrow sprite is technically the first sprite under the mouse location.

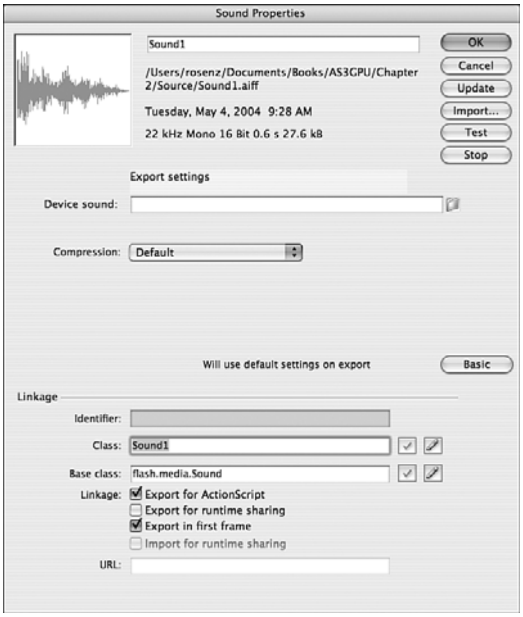
The example movie **`CustomCursor.fla`** has a simple button placed on the stage so that you can test the custom cursor rolling over a simple button.

Playing Sounds

There are two main ways to play sounds in ActionScript 3.0: as internal library sounds or as external files.

Probably the best method for most game sound effects is to embed the sounds in the game movie's library.

You can do this by importing the sound using the File, Import, Import to Library menu command. Then, after the sound is in the library, select it and look at its Sound Properties dialog. You can see an example of this dialog in Figure 2.20.



Sound1

/Users/rosenz/Documents/Books/AS3GPU/Chapter 2/Source/Sound1.aiff

Tuesday, May 4, 2004 9:28 AM

22 kHz Mono 16 Bit 0.6 s 27.6 kB

Export settings

Device sound:

Compression: Default

Will use default settings on export

Linkage

Identifier:

Class: Sound1

Base class: flash.media.Sound

Linkage: ☑ Export for ActionScript
☐ Export for runtime sharing
☑ Export in first frame
☐ Import for runtime sharing

URL:

Figure 2.20
The Sound Properties dialog lets you set the class identifier for a sound so that you can use it in ActionScript.

To use a sound in ActionScript, you need to set the linkage of the sound to Export for ActionScript, and then the class to a name you will use in your code. In this case, we'll use the name Sound1.

Then, to play the sound, you just need two lines of code:

```
var sound1:Sound1 = new Sound1();
var channel:SoundChannel = sound1.play();
```

Or, if you want to be more concise, you could do it in one line:

```
var channel:SoundChannel = (new Sound1()).play();
```

Playing an external file is a little more difficult. First, you need to load the sound into an object. This code loads the sound file **`PlayingSounds.mp3`** into the object `sound2`:

```
var sound2:Sound = new Sound();
var req:URLRequest = new URLRequest("PlayingSounds.mp3");
sound2.load(req);
```

Then, to play the sound, you just need the `play` command:

```
sound2.play();
```

The example movie **`PlayingSounds.fla`** has two buttons: one that plays a library sound, and one that plays an external sound. The external sound is loaded as soon as the movie begins, so it is ready to be played at any time.

NOTE

It is possible, with longer external sounds, that the sound wouldn't have completed loading before it is needed. You can detect this by using the `isBuffering` property of the sound object. You can also use the `bytesLoaded` and `bytesTotal` properties for more advanced tracking.

However, even if the sound is not quite finished loading, it will start to play as soon as it is loaded. So, for short sounds, it is probably not necessary to worry about it.

Loading Screen

Flash is built for streaming content. This means that the movie will start when only the bare minimum content has been loaded, such as the elements used on the first frame.

This is great for animation. You can have a 1,000-frame hand-crafted animation that will start playing immediately and continue to load elements needed for future frames as the user watches the early frames.

But for games, we rarely want to do this. Game elements are used by our ActionScript code almost immediately. If we are missing one of these elements because it hasn't been loaded yet, the game could fail to play properly.

So, most games use a loading screen that forces the movie to wait until everything is downloaded. It also keeps the player informed of the status of the download.

A simple way to do this is place a `stop` on the first frame of the movie so that the movie won't play until you tell it to:

```
stop();
```

Then, set an `ENTER FRAME` listener to call a `loadProgress` function every frame:

```
addEventListener(Event.ENTER_FRAME, loadProgress);
```

This function gets the status of the movie using `this.root.loaderInfo`. It has properties `bytesLoaded` and `bytesTotal`. We'll take these and also convert them to kilobytes by dividing by 1,024:

```
function loadProgress(event:Event) {
     // get bytes loaded and bytes total
     var movieBytesLoaded:int = this.root.loaderInfo.bytesLoaded;
     var movieBytesTotal:int = this.root.loaderInfo.bytesTotal;

     // convert to KiloBytes
     var movieKLoaded:int = movieBytesLoaded/1024;
     var movieKTotal:int = movieBytesTotal/1024;
```

To show the player the loading progress, we just put some text into a text field that is already on frame 1 of the movie. It will display text like "Loading: 5K/32K":

```
     // show progress
     progressText.text = "Loading: "+movieKLoaded+"K/"+movieKTotal+"K";
```

When the `movieBytesLoaded` equals the `movieBytesTotal`, we'll remove the event listener, and push the movie forward to frame 2. If this is the beginning of an animated sequence, you can use `gotoAndPlay` instead:

```
     // move on if done
     if (movieBytesLoaded >= movieBytesTotal) {
          removeEventListener(Event.ENTER_FRAME, loadProgress);
          gotoAndStop(2);
     }
}
```

The example movie **`LoadingScreen.fla`** contains this code in the first frame. It also has a 33K image on the second frame. To test the code, first test the movie normally using Control, Test Movie. Then, in the testing environment, choose View, Download Settings, 56K. Then choose View, Simulate Download. This will simulate loading at 4.7K/sec, and allow you to see the loading screen in action.

Random Numbers

Random numbers are used in almost any game. They allow for infinite variation and help you keep your code simple.

Creating random numbers in ActionScript 3.0 is done with the `Math.random` function. This returns a value between 0.0 and 1.0, not including 1.0 itself.

NOTE

The number returned is generated by a complex algorithm in the Flash Player. It seems to be completely random; but because it is an algorithm, it isn't technically completely random. However, for our purposes as game developers, we don't need to worry about it and can consider the numbers returned completely random.

So, this code will get a number from 0.0 to 1.0, not including 1.0:

```
var random1:Number = Math.random();
```

We'll usually want to define a more specific range for the random number. For instance, you might want a random number between 0 and 10. We define these ranges by simply multiplying the result of `Math.random` by the range:

```
var random2:Number = Math.random()*10;
```

If you want an integer value, rather than a floating-point number, you could use `Math.floor` to round the values down. This will give you a random number from 0 to 9:

```
var random3:Number = Math.floor(Math.random()*10);
```

If you want to define a range that doesn't start at 0, you just need to add to the result. This will give you a result from 1 to 10:

```
var random4:Number = Math.floor(Math.random()*10)+1;
```

The movie **`RandomNumbers.fla`** shows these lines of code, with output to the Output panel.

Shuffling an Array

One of the most common uses of random numbers in games is to set up the game pieces at the start of a game. Typically, this involves a shuffle of playing elements, such as cards, tiles, or playing pieces.

For instance, say you have 52 playing pieces that you want to shuffle into a random order, like a dealer would shuffle cards before dealing a hand of poker or blackjack.

The way this is done is to first create the array of playing pieces as a simple ordered array. The following code does this with the numbers 0 through 51:

```
// create ordered array
var startDeck:Array = new Array();
for(var cardNum:int=0;cardNum<52;cardNum++) {
    startDeck.push(cardNum);
}
trace("Unshuffled:",startDeck);
```

The result to the Output window will look like this:

```
Unshuffled:
0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,3
2,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,50,51
```

To randomize the array, we choose a random position in the array and take the number from that position and place it in a new array. Then, we delete the number from the old array. We'll keep doing this until the old array is empty:

```
// shuffle into new array
var shuffledDeck:Array = new Array();
while (startDeck.length > 0) {
     var r:int = Math.floor(Math.random()*startDeck.length);
     shuffledDeck.push(startDeck[r]);
     startDeck.splice(r,1);
}
trace("Shuffled:", shuffledDeck);
```

The result will look something like this. Of course, it will be different each time you run the program:

```
Shuffled:
3,42,40,16,41,44,30,27,33,11,50,0,21,23,49,29,20,28,22,32,39,25,17,19,8,7,10,37,2,12,31
,5,46,26,48,45,43,9,4,38,15,36,51,24,14,18,35,1,6,34,13,47
```

The example movie **`ShufflingAnArray.fla`** demonstrates this procedure.

Displaying a Clock

Keeping track of time can be done with the `getTimer()` function. This tells you the number of milliseconds that have elapsed since the Flash player started.

A typical internal game clock notes the start of game play by placing the `getTimer()` value at that moment in a variable. For instance, the game might start 7.8 seconds after the Flash Player starts, perhaps delayed that long while the user finds and clicks the "Start Game" button. So, the value `7800` will be stored in `startTime`.

Then, to get the time at any point, it is just a matter of subtracting `startTime` from the current time.

However, the player isn't interested in raw milliseconds. The player wants to see something like "1:34" for 1 minute and 34 seconds.

Converting milliseconds to a time format is just a matter of dividing by 1,000 to get the number of seconds, and then by 60 to get the number of minutes.

Here is an example of a program that places a text field on the screen, captures the time at the start, and then displays the clock every frame. It converts the time to seconds and minutes, padding a zero on to the number of seconds if it is less than ten:

```
var timeDisplay:TextField = new TextField();
addChild(timeDisplay);

var startTime:int = getTimer();
addEventListener(Event.ENTER_FRAME, showClock);

function showClock(event:Event) {
     // milliseconds passed
     var timePassed:int = getTimer()-startTime;

     // compute minutes and seconds
     var seconds:int = Math.floor(timePassed/1000);
     var minutes:int = Math.floor(seconds/60);
     seconds -= minutes*60;

     // convert to clock string
     var timeString:String = minutes+":"+String(seconds+100).substr(1,2);

     // show in text field
     timeDisplay.text = timeString;
}
```

Let's take a closer look at the string conversion. The number of minutes is taken straight from the `minutes` variable. Then, the colon is appended after it.

The number of seconds is handled differently: 100 is added to it, so 7 seconds becomes 107 seconds; 52 seconds becomes 152 seconds; and so on. Then, it is converted to a string with the `String` constructor. Then, the substring starting at character 1 and with a length of 2 is taken. Because we start counting characters at 0, this means the 07 or the 52 is taken, never the 1 at the start of 107 or 152.

The result is strings like 1:07 or 23:52. You can check the example movie **`DisplayingAClock.fla`** to see this code in action.

System Data

It is often necessary to know information about what type of computer your game is being played on. This might affect how you want your game to handle certain situations and levels of detail.

For instance, you can get the stage width and height using the `stage.stageWidth` and `stage.stageHeight` properties. These values will even change in real time if the movie is set to scale to fit the browser.

So, if your movie is built to be 640 pixels wide, but you detect that it is playing at 800 pixels wide, you can choose to show more detail so that the player can see greater detail. Or, you can choose to show fewer frames of animation, because a larger scale means more rendering power is needed.

You can also use the `Capabilities` object to get various pieces of information about the computer. Here is a handy list of the ones that will most affect us as game developers:

- **`Capabilities.playerType`**. This returns `External` if you are testing the movie, `StandAlone` if it is running as a Flash Projector, `PlugIn` if it is running in a browser like FireFox or Safari, or `ActiveX` if it is running in Internet Explorer. So, you could put some cheats into your code that only work if the `playerType` is `External` that allow you to test your game, but will not affect the web-based version.
- **`Capabilities.language`**. This returns the two-letter code, such as `en` for English, if the computer is set to use this as the primary language.
- **`Capabilities.os`**. Returns the operating system type and version, such as Mac OS 10.4.9.
- **`Capabilities.screenResolutionX`**, **`Capabilities.screenResolutionY`**. The display resolution, such as 1280 and 1024.
- **`Capabilities.version`**. The Flash Player version, such as MAC 9,0,45,0. You can extract the operating system or player version from this.

There are many more `Capabilities` properties you can grab. Check the Flash CS3 documentation. Also see **`SystemData.fla`** for a movie that grabs most of the data above and displays it live in a text field.

Game Theft and Security

Game theft on the Internet is a big problem. Most games are not protected in any way, and it is easy for someone to grab the SWF file and upload it to a website, claiming your work as their own.

There are many ways to prevent this. The simplest is to have your game check to make sure it is running from your server.

This can be done with the `this.root.loaderInfo.url` property. It returns the full path of the SWF file, starting with `http://` if the file is on the Web.

You can then check it against your domain. For instance, to make sure that `flashgameu.com` appears in the path, you could do this:

```
if (this.root.loaderInfo.url.indexOf("flashgameu.com") != -1) {
    info.text = "Is playing at flashgameu.com";
} else {
    info.text = "Is NOT playing at flashgameu.com";
}
```

Instead of simply setting a text field, you could stop the game from playing, or send the player to your site with `navigateToURL`.

After you have secured your game at your site, the next step is to secure it so that someone can't use an `EMBED` tag with an absolute URL to your SWF file. With this method, they are embedding your game from your server into their web page on their server.

There is no easy way to stop this. However, if you discover it happening, you can always move your SWF file. In fact, you can even replace your SWF file with one that does nothing but redirect the player using `navigateToURL`.

NOTE
Some web servers can prevent remote linking. This is mostly used to prevent people from embedding images from your server into their pages. In many cases, it also works with SWF files. You should check with your ISP about this functionality.

A more advanced method to prevent embedded linking is relatively complex. Basically, it involves passing in a secret value to the Flash movie by two alternate routes: as a `flashvars` parameter, and as a bit of text or XML data using `URLLoader`. If the two secret values don't match, the Flash movie must be stolen.

The idea is to change the value passed in by both methods on a regular basis. So, if someone steals your SWF movie but does not take your HTML code to embed the Flash movie into the page, your movie won't get the `flashvars` version of the secret value, and so won't work for that person.

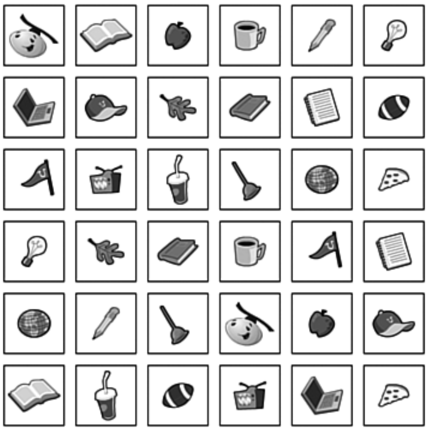
Even if they steal your HTML code, they will have only the current version of the `flashvars` secret value. For the time being, it will match the `URLLoader` secret value. After you update the secret value in both places, the old `flashvars` secret value on the thief's page will cease to match the new `URLLoader` value on your server.

Of course, it is still possible for someone to steal your SWF game, open it with a SWF Decompiler program, and remove your security. So, there is no 100 percent solution.

However, most thieves will look for games that are easy to steal. Don't make yours one of them.

Now that you've learned some more ActionScript 3.0 programming techniques through these small code building blocks, it is time to move on and make our first game.

This page intentionally left blank



3

Basic Game Framework: A Matching Game

- Placing Interactive Elements
- Game Play
- Encapsulating the Game
- Adding Scoring and a Clock
- Adding Game Effects
- Modifying the Game

SOURCE FILES
http://flashgameu.com

A3GPU03_MatchingGame.zip

To build our first game, I've chosen one of the most popular games you will find on the Web and in interactive and educational software: a matching game.

Matching games are simple memory games played in the physical world using a simple deck of cards with pictures on them. The idea is to place pairs of cards face down in a random arrangement. Then, try to find matches by turning over two cards at a time. When the two cards match, they are removed. If they don't match, they are turned face down again.

A good player is one who remembers what cards he or she sees when a match is not made, and can determine where pairs are located after several failed tries.

NOTE

Sometimes educational matching games for children don't have exact matches in a pair of cards. For instance, one card can have a picture of a cat on it, whereas the other card has the word *cat*. Or, one card can have the number 7, whereas another has 3+4.

Computer versions of matching games have advantages over physical versions: You don't need to collect, shuffle, and place the cards to start each game. The computer does that for you. It is also easier and less expensive for the game developer to create different pictures for the cards with virtual cards rather than physical ones.

To create a matching game, we first work on placing the cards on the screen. To do this, we need to shuffle the deck to place the cards in a random order each time the game is played.

Then, we take the player's input and use that to reveal the pictures on a pair of cards. Then, we compare the cards and remove them if they match.

We also need to turn cards back to their face-down positions when a match is not found. And then we need to check to see when all the pairs have been found so that the game can end.

Placing Interactive Elements

Creating a matching game first requires that you create a set of cards. Because the cards need to be in pairs, we need to figure out how many cards will be displayed on the screen, and make half that many pictures.

For instance, if we want to show 36 cards in the game, there will be 18 pictures, each appearing on 2 cards.

Methods for Creating Game Pieces

There are two schools of thought when it comes to making game pieces, like the cards in the matching game.

Multiple-Symbol Method

The first method is to create each card as its own movie clip. So, in this case, there will be 18 symbols. Each symbol represents a card.

One problem with this method is that you will likely be duplicating graphics inside of each symbol. For instance, each card would have the same border and background. So, you would have 18 copies of the border and background.

Of course, you can get around this by creating a background symbol that is then used in each of the 18 card symbols.

NOTE

Using multiple symbols, one for each card, can prove useful if you are picking cards from a large group—like if you need 18 cards from a pool of 100. Or, it could be useful if the cards are being imported into the movie from external media files, like a bunch of JPG images.

But the multiple-symbol method still has problems when it comes to making changes. For instance, suppose you want to resize the pictures slightly. You'd need to do that 18 times for 18 different symbols.

Also, if you are a programmer teaming up with an artist, it is inconvenient to have the artist update 18 or more symbols. If the artist is a contractor, it could run up the budget as well.

Single-Symbol Method

The second method for working with a set of playing pieces, such as cards, is a single-symbol method. You would have one symbol, a movie clip, with multiple frames. Each frame contains the graphics for a different card. Shared graphics, such as a border or background, can be on a layer in the movie clip that stretches across all the frames.

NOTE

Even the single-symbol method can use many symbols. For instance, if your playing pieces are deck of poker cards, you may place the four suits (spades, hearts, diamonds, and clubs) in symbols and use them in your main deck symbol on the cards. That way, if you want to change how the hearts look across your whole deck, you can do this by just changing the heart symbol.

This method has major advantages when it comes to updates and changes to the playing pieces. You can quickly and easily move between and edit all the frames in the movie clip. You can also easily grab an updated movie clip from an artist with whom you are working.

Setting Up the Flash Movie

Using the single-symbol method, we need to have at least one movie clip in the library. This movie clip will contain all the cards, and even a frame that represents the back of the card that we must show when the card is face down.

Create a new movie that contains a single movie clip called `Cards`. To create a new movie in Flash CS3, choose File, New, and then you will be presented with a list of file types. You must choose Flash File (ActionScript 3.0) to create a movie file that will work with the ActionScript 3.0 class file we are about to create.

Put at least 19 frames in that movie clip, representing the card back and 18 card fronts with different pictures on them. You can open the **MatchingGame1.fla** file for this exercise if you don't have your own symbol file to use.

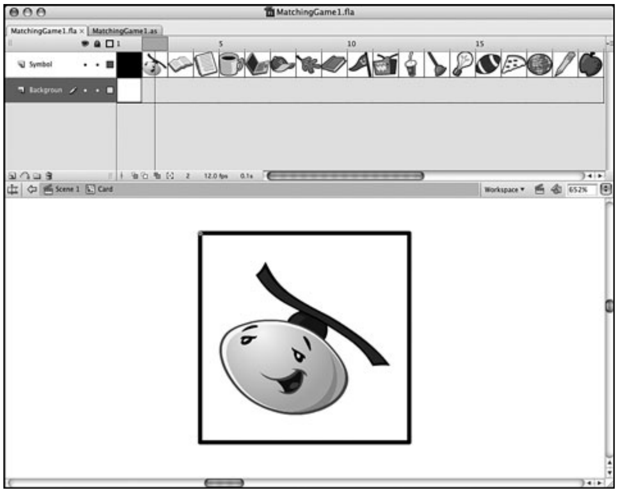
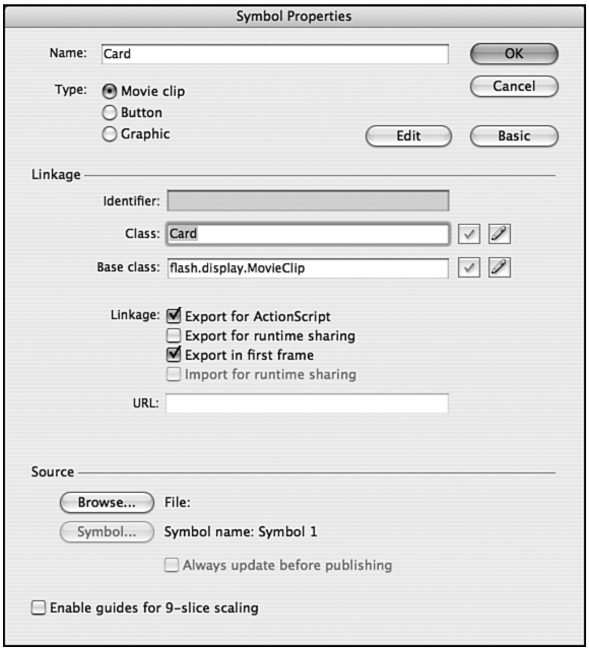


Figure 3.1
The `Card` movie clip is a symbol with 37 frames. Each frame represents a different card.

Figure 3.1 shows a timeline for the `Card` movie clip we will be using in this game. The first frame is "back" of the card. It is what the player will see when the card is supposed to be face down. Then, each of the other frames shows a different picture for the front of a card.

After we have a symbol in the library, we need to set it up so that we can use it with our ActionScript code. To do this, we need to set its properties by selecting it in the library and bringing up the Symbol Properties dialog box (see Figure 3.2).



Symbol Properties

Name: Card

Type: [✓ Movie clip] Button Graphic

Linkage

Identifier:

Class: Card

Base class: flash.display.MovieClip

Linkage: [✓] Export for ActionScript
[ ] Export for runtime sharing
[✓] Export in first frame
[ ] Import for runtime sharing

URL:

Source

File:

Symbol name: Symbol 1

[ ] Always update before publishing

[ ] Enable guides for 9-slice scaling

Figure 3.2
The Symbol Properties dialog box shows the properties for the symbol Card.

Set the symbol name to **`Card`** and its type to Movie Clip. For ActionScript to be able to work with the Cards movie clip, it needs to be assigned a class. By checking the Export for ActionScript box, we automatically get the class name **`Card`** assigned to the symbol. This will be fine for our needs here.

There is nothing else needed in the Flash movie at all. The main timeline is completely empty. The library has only one movie clip in it, the Cards movie clip. All that we need now is some ActionScript.

Creating the Basic ActionScript Class

To create an ActionScript class file, choose File, New, and then select ActionScript File from the list of file types; by doing so you create an untitled ActionScript document that you can type into.

We start off an ActionScript 3.0 file by defining it as a `package`. This is done in the first line, as you can see in the following code sample:

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

Right after the package declaration, we need to tell the Flash playback engine what classes we need to accomplish our tasks. In this case, we go ahead and tell it we'll be needing access to the entire `flash.display` class and all its immediate subclasses. This will give us the ability to create and manipulate movie clips like the cards.

The class declaration is next. The name of the class must match the name of the file exactly. In this case, we call it `MatchingGame1`. We also need to define what this class will affect. In this case, it will affect the main `Flash` movie, which is a movie clip:

```
public class MatchingGame1 extends MovieClip {
```

Next is the declaration of any variables that will be used throughout the class. However, our first task of creating the 36 cards on the screen is so simple that we don't need to use any variables. At least not yet.

Therefore, we can move right on to the initialization function, also called the *constructor* function. This function runs as soon as the class is created when the movie is played. It must have exactly the same name as the class and the ActionScript file:

```
public function MatchingGame1():void {
```

This function does not need to return any value, so we can put `:void` after it to tell Flash that nothing will ever be returned from this function. We can also leave the `:void` off, and it will be assumed by the Flash compiler.

Inside the constructor function we can perform the task of creating the 36 cards on the screen. We'll make it a grid of 6 cards across by 6 cards down.

To do this, we use two nested `for` loops. The first moves the variable `x` from 0 to 5. The `x` will represent the column in our 6x6 grid. Then, the second loop will move `y` from 0 to 5, which will represent the row:

```
                    for(var x:uint=0;x<6;x++) {
                            for(var y:uint=0;y<6;y++) {
```

Each of these two variables is declared as a `uint`, an unsigned integer, right inside the `for` statement. Each will start with the value 0, and then continue while the value is less than 6. And, they will increase by one each time through the loop.

NOTE

There are three types of numbers: `uint`, `int`, and `Number`. The `uint` type is for whole numbers 0 or higher. The `int` type is for whole numbers that can be positive or negative. The `Number` type can be positive or negative numbers, whole or floating point, such as 3.5 or –173.98. In `for` loops, we usually use either `uint` or `int` types because we only move in whole steps.

So, this is basically a quick way to loop and get the chance to create 36 different Card movie clips. Creating the movie clips is just a matter of using `new`, plus `addChild`. We also want to make sure that as each new movie clip is created it is stopped on its first frame and is positioned on the screen correctly:

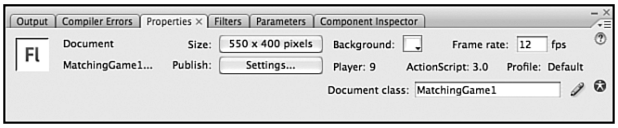
```
                    var thisCard:Card = new Card();
                    thisCard.stop();
                    thisCard.x = x*52+120;
                    thisCard.y = y*52+45;
                    addChild(thisCard);
                }
            }
        }

    }
}
```

NOTE

Adding a symbol in ActionScript 3.0 takes only two commands: `new`, which allows you to create a new instance of the symbol; and `addChild`, which adds the instance to the display list for the stage. In between these two commands, you want to do things such as set the `x` and `y` position of the new symbol.

The positioning is based on the width and height of the cards we created. In the example movie **MatchingGame1.fla**, the cards are 50 by 50 with 2 pixels in between. So, by multiplying the `x` and `y` values by 52, we space the cards with a little extra space between each one. We also add 120 horizontally and 45 vertically, which happens to place the card about in the center of a 550x400 standard Flash movie.

Before we can test this code, we need to link the Flash movie to the ActionScript file. The ActionScript file should be saved as **MatchingGame1.as**, and located in the same directory as the **MatchingGame1.fla** movie.



Document
MatchingGame1...
Size: 550 x 400 pixels
Publish: Settings...
Background:
Frame rate: 12 fps
Player: 9
ActionScript: 3.0
Profile: Default
Document class: MatchingGame1

Figure 3.3
You need to set the Document class of a Flash movie to the name of the AS file that contains your main script.

However, that is not all you need to do to link the two. You also need to set the Flash movie's Document class property in the Property Inspector. Just select the Properties tab of the Property Inspector while the Flash movie **MatchingGame1.fla** is the current document. Figure 3.3 shows the Property Inspector, and you can see the Document class field at the bottom right.

NOTE

You can test a movie when either the Flash movie itself is the current document or an ActionScript file is the current document. When an ActionScript file is the current document, look in the upper-right part of the document window for a Target indicator. This tells you what Flash movie will be compiled and run when you test. If the wrong file is shown as the Target, you can use the drop-down menu to change it.

Figure 3.4 shows the screen after we have tested the movie. The easiest way to test is to go to the menu and choose Control, Test Movie.

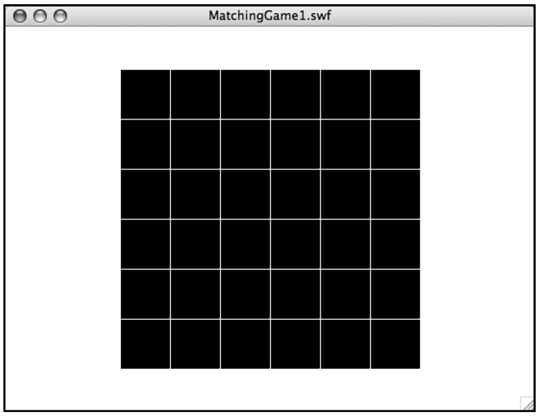


Figure 3.4
The screen shows 36 cards, spaced and in the center of the stage.

Using Constants for Better Coding

Before we go any further with developing this game, let's look at how we can make what we have better. We'll copy the existing movie to **MatchingGame2.fla** and the code to **MatchingGame2.as**. Remember to change the document class of **MatchingGame2.fla** to MatchingGame2 and the class declaration and constructor function to `MatchingGame2`.

Suppose you don't want a 6x6 grid of cards. Maybe you want a simpler 4x4 grid. Or even a rectangular 6x5 grid. To do that, you just need to find the `for` loops in the previous code and change the loops so that they loop with different amounts.

A better way to do it is to remove the specific numbers from the code all together. Instead, have them at the top of your code, and clearly labeled, so that you can easily find and change them later on.

NOTE

Putting specific numbers in your code, such as the 6s for the row and column lengths, is called hard coding. It is considered to be a bad practice for programmers because it makes it harder to adjust your program later, and especially hard for others to inherit your code and find out where they can adjust it.

We've got several other hard-coded values in our programs. Let's make a list.

Horizontal Rows = 6
Vertical Rows = 6
Horizontal Spacing = 52
Vertical Spacing = 52
Horizontal Screen Offset = 120
Vertical Screen Offset = 45

Instead of placing these values in the code, let's put them in some constant variables up in our class, to make them easy to find and modify:

```
public class MatchingGame2 extends MovieClip {
    // game constants
    private static const boardWidth:uint = 6;
    private static const boardHeight:uint = 6;
    private static const cardHorizontalSpacing:Number = 52;
    private static const cardVerticalSpacing:Number = 52;
    private static const boardOffsetX:Number = 120;
    private static const boardOffsetY:Number = 45;
```

NOTE

Notice that I chose `private static const` when defining each constant. The `private` means that these variables can only be accessed inside this class. The `static` means that they will have the same values in all instances of the class. And, the `const` means that the values can never change. If you were to use `public var` instead, it would give you the opposite declaration: can be accessed outside of the class, and will hold different values for each instance. Because this is the one and only instance of the class, and there are no outside scripts, it really makes no difference, except for neatness.

Now that we have constants, we can replace the code in the constructor function to use them rather than the hard-coded numbers:

```
public function MatchingGame2():void {
    for(var x:uint=0;x<boardWidth;x++) {
        for(var y:uint=0;y<boardHeight;y++) {
            var thisCard:Card = new Card();
            thisCard.stop();
            thisCard.x = x*cardHorizontalSpacing+boardOffsetX;
```

```
                thisCard.y = y*cardVerticalSpacing+boardOffsetY;
                addChild(thisCard);
            }
        }
    }
```

You can see that I also changed the name of the class and function to MatchingGame2. You can find these in the sample files **MatchingGame2.fla** and **MatchingGame2.as**.

NOTE

As we move through this chapter, we'll be changing the filenames of both the ActionScript file and the movie. If you are following along by creating your own movies from scratch, also remember to change the document class in the Property Inspector so each movie points to the right ActionScript file. For instance, the **MatchingGame2.fla** movie needs to use the **MatchingGame2.as** file, so its document class should be set to MatchingGame2.

In fact, open those two files. Test them one time. Then, test them again after you change some of the constants. Make the `boardHeight` only five cards, for instance. Scoot the cards down by 20 pixels by changing `boardOffsetY`. The fact that you can make these changes quickly and painlessly drives home the point of using constants.

Shuffling and Assigning Cards

Now that we can add cards to the screen, we want to assign the pictures randomly to each card. So, if there are 36 cards in the screen, there should be 18 pairs of pictures in random positions.

Chapter 2, "ActionScript Game Elements," discussed how to use random numbers. However, we can't just pick a random picture for each card. We need to make sure there are exactly two of each type of card on the screen. No more, no less; otherwise, there will not be matching pairs.

NOTE

This process is kind of the opposite from shuffling a deck of cards. Instead of mixing the cards and then picking new cards from the top of the deck, we'll be using an ordered list of cards and picking new cards from random spots in the deck.

To do this, we need to create an array that lists each card, and then pick a random card from this array. The array will be 36 items in length, containing 2 of each of the 18 cards. Then, as we create the 6x6 board, we'll be removing cards from the array and placing them on the board. When we have finished, the array will be empty, and all 18 pairs of cards will be accounted for on the game board.

Here is the code to do this. A variable `i` is declared in the `for` statement. It will go from zero to the number of cards needed. This is simply the board width times the board height, divided by two (because there are two of each card). So, for a 6x6 board, there will be 36 cards. We must loop 18 times to add 18 pairs of cards:

```
// make a list of card numbers
var cardlist:Array = new Array();
for(var i:uint=0;i<boardWidth*boardHeight/2;i++) {
     cardlist.push(i);
     cardlist.push(i);
}
```

The `push` command is used to place a number in the array, twice. Here is what the array will look like:

```
0,0,1,1,2,2,3,3,4,4,5,5,6,6,7,7,8,8,9,9,10,10,11,11,12,12,13,13,14,14,15,15,16,16,17,17
```

Now as we loop to create the 36 movie clips, we'll pull a random number from this list to determine which picture will display on each card:

```
for(var x:uint=0;x<boardWidth;x++) { // horizontal
     for(var y:uint=0;y<boardHeight;y++) { // vertical
          var c:Card = new Card(); // copy the movie clip
          c.stop(); // stop on first frame
          c.x = x*cardHorizontalSpacing+boardOffsetX; // set position
          c.y = y*cardVerticalSpacing+boardOffsetY;
          var r:uint = Math.floor(Math.random()*cardlist.length); // get a random face
          c.cardface = cardlist[r]; // assign face to card
          cardlist.splice(r,1); // remove face from list
          c.gotoAndStop(c.cardface+2);
          addChild(c); // show the card
     }
}
```

The new lines are in the middle of the code. First, we use this line to get a random number between zero and the number of items remaining in the list:

```
var r:uint = Math.floor(Math.random()*cardlist.length);
```

The `Math.random()` function will return a number from 0.0 up to just before 1.0. Multiply this by `cardlist.length` to get a random number from 0.0 up to 35.9999. Then use `Math.floor()` to round that number down so that it is a whole number from 0 to 35—that is, of course, when there are 36 items in the `cardlist` array at the start of the loops.

Then, the number at the location in `cardlist` is assigned to a property of u named `cardface`. Then, we use the `splice` command to remove that number from the array so that it won't be used again.

NOTE

Although we usually need to declare and define variables, we can also add dynamic properties such as `cardface` to an object. This can only be done if the object is dynamic, which the `Card` object is by default because we did not define it otherwise. The `cardface` property will assume the type of the value it is assigned (such as a `Number`, in this case).

This is not the best programming practice. Better would be to define a class for the Card, complete with an ActionScript file declaring a package, class, properties, and constructor function. However, this is quite a lot of extra work when only one little property is needed, so the benefits of convenience outweigh the benefits of sticking to strict programming practices.

In addition, the **MatchingGame3.as** script includes this line to test that everything is working so far:

```
c.gotoAndStop(c.cardface+2);
```

This syntax makes the `Card` movie clip show its picture. So, all 36 cards will be face up rather than face down. It takes the value of the property `cardface`, which is a number from 0 to 17, and then adds 2 to get a number from 2 to 19. This corresponds to the frames in the `Card` movie clip, where frame 1 is the back of the card, and frames 2 and so on are the picture faces of the cards.

Obviously, we don't want to have this line of code in our final game, but it is useful at this point to illustrate what we have accomplished. Figure 3.5 shows what the screen might look like after we run the program with this testing line in place.

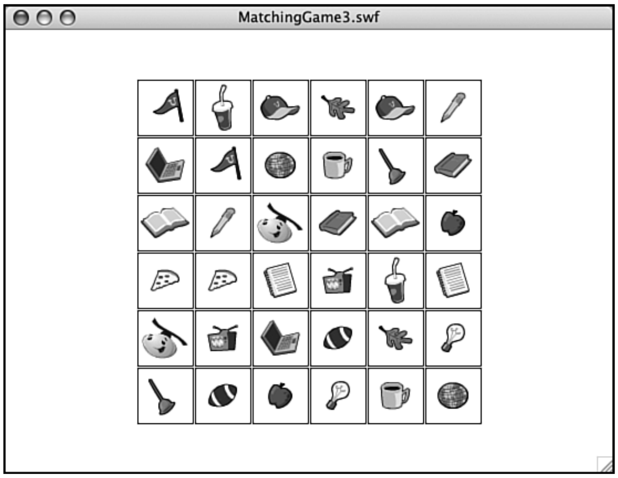


Figure 3.5
The third version of our program includes code that reveals each of the cards. This is useful to get visual confirmation that your code is working so far.

Game Play

Now that the game board is set up, we need to let the user click cards to try to find matches. We also need to keep track of play state, which in this case means whether the player is clicking the first card or second card, and whether all the cards have been found.

Adding Keyboard Listeners

The first step is to get each of the cards we create to respond to mouse clicks. We can do this by adding a listener to each of these objects. The `addEventListener` function will do this, and it takes two parameters: which event to listen for, and what function to call when the event occurs. Here is the line of code:

```
c.addEventListener(MouseEvent.CLICK,clickCard);
```

You also need to add another `import` statement at the start of the class to tell Flash you want to use events:

```
import flash.events.*;
```

The syntax for the event in this case is `MouseEvent.CLICK`, which is just a simple click on the card. When this happens, it should call the function `clickCard`, which we have yet to create. We need to create it before testing the movie again because Flash won't compile our movie with a loose end.

Here is a simple start to the `clickCard` function:

```
public function clickCard(event:MouseEvent) {
     var thisCard:Card = (event.currentTarget as Card); // what card?
     trace(thisCard.cardface);
}
```

NOTE

Using a `trace` statement call to check your code is a great way to program in small steps to avoid headaches. For instance, if you add 27 lines of code at once and then the program doesn't work as expected, you must locate the problem in 27 new lines of code. If you add only five new lines of code, however, and then use a `trace` statement to display the values of key variables, you can solve any problems with those five lines of code before moving on.

Any time you have a function that responds to an event, it must take at least one parameter, the event itself. In this case, it is a value of type `MouseEvent`, which we will assign to the variable event.

NOTE

You need to accept the event parameter on an event listener function whether you care about its value or not. For instance, if you create a single button and know that the function will only run when that button is pressed, you still need to accept the event as a parameter and then just not use it for anything.

In this case, the `event` parameter is key because we need to know which of the 36 cards the player clicked. The `event` parameter value is actually an object with all sorts of properties, but the only property we need to know about is which `Card` object was clicked. This would be the target, or more precisely, the `currentTarget` of the event.

However, the `currentTarget` is a vague object to the ActionScript engine at this point. Sure, it is a `Card` object. However, it is also a movie clip, which is a display object, too. We want to get its value as a `Card` object, so we define a variable as a `Card`, and then use a `Card` to specify that we want the value of `event.currentTarget` to be returned as a Card.

Now that we have a `Card` object in the variable `thisCard`, we can access its `cardface` property. We'll use `trace` to put it in the Output window and run a quick test of **MatchingGame4.fla** to make sure it is working.

Setting Up Game Logic

When a player clicks a card, we need to determine what steps to take based on their choice and the state of the game. There are three main states we need to deal with:

- **State 1.** No cards have been chosen, player selects first card in a potential match.
- **State 2.** One card has been chosen, player selects a second card. A comparison must be made and action taken based on whether there is a match.
- **State 3.** Two cards have been chosen, but no match was found. Leave those cards face up until a new card is chosen, and then turn them both over and reveal the new card.

Figures 3.6 through 3.8 show the three game states.

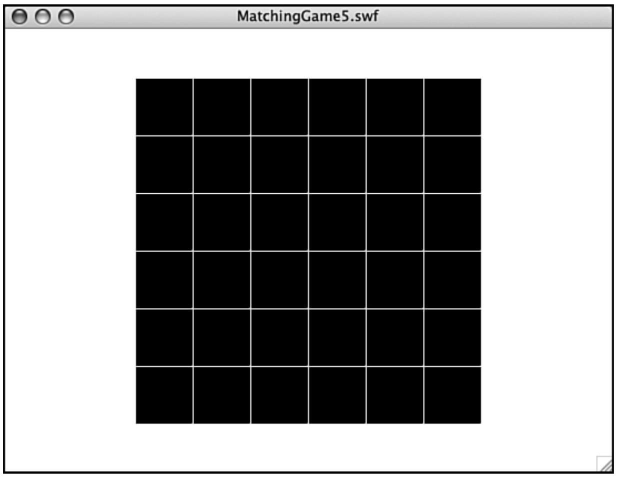


Figure 3.6
State 1, where the user is about to choose his or her first card.

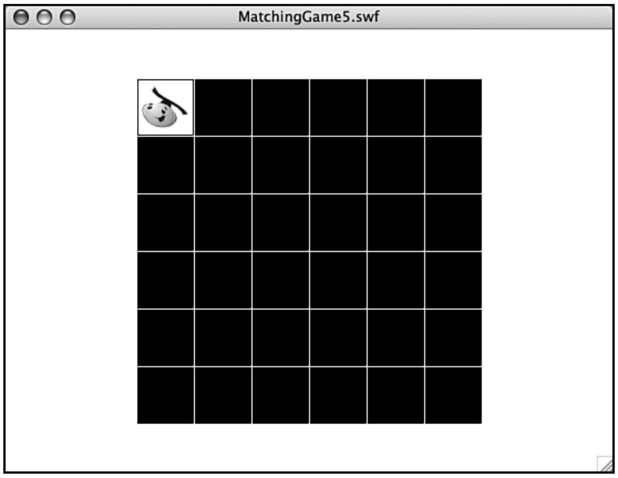


Figure 3.7
State 2, where the user is about to choose his or her second card.

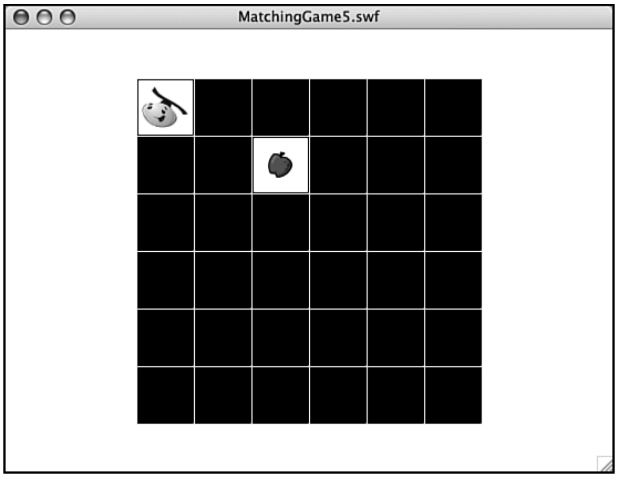


Figure 3.8
State 3, where a pair of cards was selected, but no match found. Now the user must choose another card to start a second pair.

Then, there are some other considerations. What if the player clicks a card, and then clicks the same card again. This means the player probably wants to take back the first choice, so we should turn that card over and return to the first state.

We can predict that we will need to keep track of which cards are chosen when the player is going for a match. So, we need to create our first class variables. We'll call them `firstCard` and `secondCard`. They will both be of type Card:

```
private var firstCard:Card;
private var secondCard:Card;
```

Because we haven't set any values for these variables, they will both start off with the default object value of `null`. In fact, we'll use the `null` values of these two variables to determine the state.

NOTE

Not all types of variables can be set to `null`. For instance, an `int` variable will be set to zero when it is first created, unless you specify otherwise. You can't set it to `null` even if you want to.

If both `firstCard` and `secondCard` are `null`, we must be at the first state. The player is about to choose his first card.

If `firstCard` is not `null`, and `secondCard` is `null`, we are at the second state. The player will soon choose the card that he hopes matches the first.

If both `firstCard` and `secondCard` are not `null`, we are in the third state. We'll use the values of `firstCard` and `secondCard` to know which two cards to turn face down when the user chooses the next `firstCard`.

Let's have a look at the code:

```
public function clickCard(event:MouseEvent) {
     var thisCard:Card = (event.target as Card); // what card?

     if (firstCard == null) { // first card in a pair
          firstCard = thisCard; // note it
          firstCard.gotoAndStop(thisCard.cardface+2); // turn it over
```

So far, we can see what happens when the player clicks the first card. Notice that the `gotoAndStop` command is similar to the one we used to test the card shuffle earlier in the chapter. It must add 2 to the frame number so that the card values of 0 to 17 match up with the frame numbers of 2 to 19 that contain the 18 card faces.

Now that we have the value of `firstCard` set, we can expect the second click. This is handled by the next two parts of the `if` statement. This part handles the case of when the player clicks the first card again, and will turn it back over and set the value of `firstCard` back to `null`:

```
    } else if (firstCard == thisCard) { // clicked first card again
        firstCard.gotoAndStop(1); // turn back over
        firstCard = null;
```

If the player clicks a different card for the second card, a comparison must be made between the two cards. We're not comparing the cards themselves, but the `cardface` property of the cards. If the faces are the same, a match has been found:

```
    } else if (secondCard == null) { // second card in a pair
        secondCard = thisCard; // note it
        secondCard.gotoAndStop(thisCard.cardface+2); // turn it over

        // compare two cards
        if (firstCard.cardface == secondCard.cardface) {
```

If a match has been found, we want to remove the cards and reset the `firstCard` and `secondCard` variables: This is done by using the `removeChild` command, which is the opposite of `addChild`. It will take the object out of the display list and remove it from view. But they are still stored in variables in this case, so we must set those to `null` so the objects are disposed by the Flash player.

```
            // remove a match
            removeChild(firstCard);
            removeChild(secondCard);
            // reset selection
            firstCard = null;
            secondCard = null;
        }
```

The next case is what happens if the player has selected a `firstCard`, but then selects a second card that doesn't match. And now goes on to click yet another card. This should turn over the first two cards back to their face-down position, which is frame 1 of the `Card` movie clip.

Immediately following that, it should set the `firstCard` to the new card, and show its picture:

```
    } else { // starting to pick another pair
        // reset previous pair
        firstCard.gotoAndStop(1);
        secondCard.gotoAndStop(1);
        secondCard = null;
        // select first card in next pair
        firstCard = thisCard;
        firstCard.gotoAndStop(thisCard.cardface+2);
    }
}
```

That's actually it for the basic game. You can test out **MatchingGame5.fla** and **MatchingGame5.as** to play it. You can select pairs of cards and see matches removed from the board.

You can consider this a complete game. You could easily stick a picture behind the cards in the main movie timeline and have the reward for winning simply be the revelation of the full picture. As an extra add-on to a website, it will work fine. However, we can go much further and add more features.

Checking for Game Over

It is likely that you will want to check for a game over state so that you can reward players with a screen telling them that they have completed the game. The game over state will be achieved when all the cards have been removed.

NOTE

In the examples in this chapter we just take the player to a screen that displays the words *Game Over.* However, you could show them an animation, or take them to a new web page, too. But we'll stick to the game programming here.

There are many ways to do this. For instance, you could have a new variable where you keep track of the number of pairs found. Every time you find a pair, increase this value by one, and then check to see when it is equal to the total number of pairs.

Another method would be to check the `numChildren` property of the `MatchingGame` object. When you add 36 cards to it, `numChildren` will be 36. As pairs get removed, `numChildren` goes to zero. When it gets to zero, the game is over.

The problem with that method is that if you place more items on the stage, such as a background or title bar, they will also be counted in `numChildren`.

In this case, I like a variation on the first idea. Instead of counting the number of cards removed, count the number of cards shown. So, create a new class variable named `cardsLeft`:

```
private var cardsLeft:uint;
```

Then, set it to zero just before the `for` loops that create the cards. And, add one to this variable for every card created:

```
cardsLeft = 0;
for(var x:uint=0;x<boardWidth;x++) { // horizontal
     for(var y:uint=0;y<boardHeight;y++) { // vertical
          var c:Card = new Card(); // copy the movie clip
          c.stop(); // stop on first frame
          c.x = x*cardHorizontalSpacing+boardOffsetX; // set position
          c.y = y*cardVerticalSpacing+boardOffsetY;
          var r:uint = Math.floor(Math.random()*cardlist.length); // get a random face
```

```
            c.cardface = cardlist[r]; // assign face to card
            cardlist.splice(r,1); // remove face from list
            c.addEventListener(MouseEvent.CLICK,clickCard); // have it listen for clicks
            addChild(c); // show the card
            cardsLeft++;
        }
    }
```

Then, in the `clickCard` function, we need to add new code when the user makes a match and the cards are removed from the screen: This goes in the `clickCard` function.

```
cardsLeft -= 2;
if (cardsLeft == 0) {
    gotoAndStop("gameover");
}
```

NOTE

You can use ++ to add one to a variable, — to subtract one. For instance, `cardsLeft++` is the same as writing `cardsLeft = cardsLeft + 1`.

You can also use += to add a number to a variable, and -= to subtract a number. For instance, `cardsLeft -= 2` is the same as writing `cardsLeft = cardsLeft - 2`.

That is all we needed for coding. Now, the game tracks the number of cards on the screen using the `cardsLeft` variable, and it will take an action when that number hits zero.

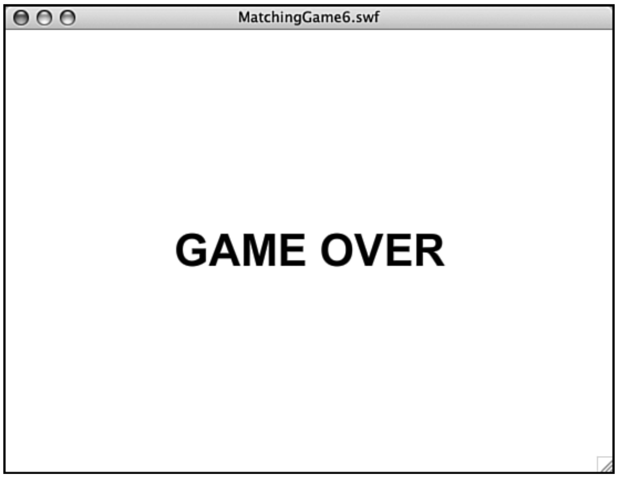


Figure 3.9
The simplest gameover screen ever.

The action it takes is to jump to a new frame, like the one shown in Figure 3.9. If you look at the movie **MatchingGame6.fla**, you can see that I added a second frame. I also added `stop();` commands to the first frame. This makes the movie stop on the first frame so that the user can play the game, instead of continuing on to the second frame. The second frame is labeled gameover and will be used when the `cardsLeft` property is zero.

At this point, we want to remove any game elements created by the code. However, because the game only creates the 36 cards, and then all 36 are removed when the player finds all the matches, there are no extra items on the screen to remove. We can jump to the gameover frame without any items on the screen at all.

The gameover screen shows the words *Game Over* in the sample movie. You can add additional graphics or even animation here, too. Later in this chapter, we look at how to add a Play Again button to this frame.

Encapsulating the Game

At this point, we have a game that runs as a whole Flash movie. The movie is **MatchingGameX.fla**, and the ActionScript class is **MatchingGameX.as**. When the movie runs, the game initializes and starts. The movie is the game, and the game is the movie.

This works well in simple situations. In the real world, however, you want to have introduction screens, gameover screens, loading screens, and so on. You might even want to have different screens with different versions of the game, or different games completely.

Flash is great at encapsulation. A `Flash` movie is a movie clip. You can have movie clips inside of movie clips. So, a game can be the movie, or a game can be a movie clip inside the movie.

Why would you want to do this? Well, for one thing, it makes it easy to add other screens to your game. So, we can make frame 1 an introduction screen, frame 2 the game, and frame 3 the gameover screen. Frame 2 would actually contain a movie clip called `MatchingGameObject7` that uses the class **MatchingGameObject7.as**.

Figure 3.10 shows a diagram of the three frames we plan to have in our updated movie, and what each one contains.

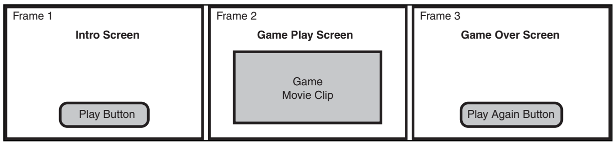
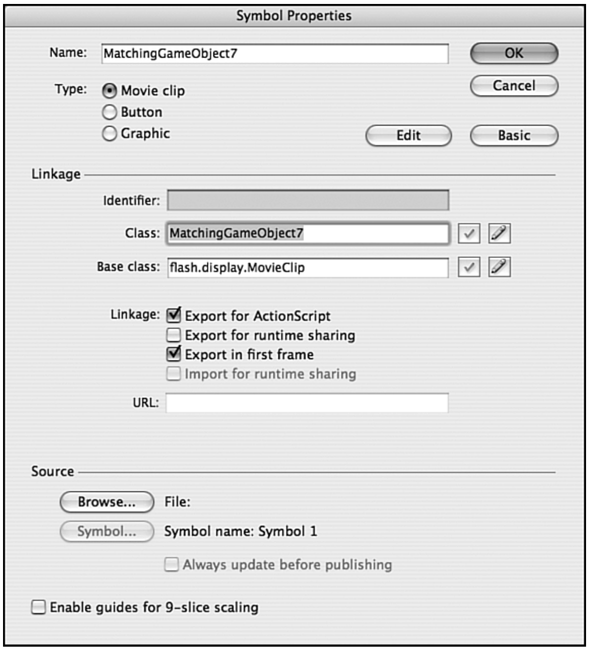


Figure 3.10
The second frame of the movie contains a movie clip, which is the actual game. The other frames contain supporting material.

Creating the Game Movie Clip

In **MatchingGameObject7.fla**, there are three frames. Let's skip right to the second frame. There, we can see a single movie clip. You might not even notice it at first because it is a completely empty movie clip and so appears as a small circle at the upper-left corner of the screen.

In the library, this movie clip is named `MatchingGameObject7`; and as shown in Figure 3.11, it is assigned the class **`MatchingGameObject7`**.



Symbol Properties

Name: MatchingGameObject7

Type: [✓ Movie clip] Button Graphic

Linkage

Identifier:

Class: MatchingGameObject7

Base class: flash.display.MovieClip

Linkage: [✓ Export for ActionScript] Export for runtime sharing [✓ Export in first frame] Import for runtime sharing

URL:

Source

File:

Symbol name: Symbol 1

Always update before publishing

Enable guides for 9-slice scaling

Figure 3.11
*This movie clip will use the **Matching GameObject7.as** file as its class.*

So, essentially, this movie clip takes over the entire game, and the main movie timeline is now a larger movie clip wrapped around it.

When the movie gets to frame 2, the `MatchingGameObject7` movie clip springs into existence, runs the class constructor function in its **MatchingGameObject7.as** class, and the game plays inside this movie clip.

When the movie goes on to frame 3, the whole game will disappear because the movie clip only exists on frame 2.

This enables us to put frames before and after the game (and thus leave the game code alone to just worry about the game).

Adding an Introduction Screen

Most games would have an introduction screen. After all, we don't want to throw players right into the game. They may need an introduction or instructions.

The intro screen will contain some scripting on the main timeline, in frame 1. First, it must stop the movie so that it doesn't continue past frame 1. Then, it should set up a button to allow users to start the game.

NOTE

If you want to keep all code off of the main timeline, you could set up a new AS class file to be the document class for the whole movie. It would run on frame 1, and you could do the same sorts of things in this class file as you could on the timeline. However, it is irresistibly easy to just add this little bit of code to the main timeline and avoid creating more files than necessary.

The frame script first needs to assign a listener to a button we will create on the first frame. We'll assign the name `playButton` to that button.

The event listener will call the function `startGame`, which simply issues a `gotoAndStop` command to the main timeline, telling it to go to the frame called playgame, which is frame 2.

We will also put a `stop` command on the frame so that when the movie runs, it will stop on frame 1 and wait for the user to click this button:

```
playButton.addEventListener(MouseEvent.CLICK,startGame);

function startGame(event:MouseEvent) {
    gotoAndStop("playgame");
}

stop();
```

On the second frame, the empty movie clip `MatchingGameObject7` will sit. Then, we need to rename the document class AS file to **MatchingGameObject7.as** so that it will be used by this movie clip and not the main movie.

NOTE

To create an empty movie clip, just go to the library and choose New Symbol for its top menu. Name the symbol, set its type to Movie Clip, and set its properties. Then, drag the movie clip from the library to the stage. Place it at the very upper-left corner so that its 0,0 location is the same as the stage's 0,0 location.

We need to make one change in the code. There is a reference to the main timeline when the game is over. The `gotoAndStop` command will no longer work properly because the game is taking place in the movie clip and the gameover frame is on the main timeline. We just need to change this as follows:

```
MovieClip(root).gotoAndStop("gameover");
```

NOTE

You would think that you could simply program `root.gotoAndStop("gameover")`. After all, `root` is indeed the main timeline and the parent of the movie clip. However, the strict ActionScript compiler will not allow it. The `gotoAndStop` command can be issued only to movie clips, and technically, `root` can be other things, such as a single-frame movie clip called a sprite. So to ensure the compiler that `root` is a movie clip, we type it using the `MovieClip()` function.

The gameover frame of the movie is the same, for the time being, as in **MatchingGame6.fla**. It is just a frame with the words *Game Over* on it.

The **MatchingGame7.fla** movie is a little different from the preceding six versions in that it doesn't have a document class assigned to it. In fact, there is no **MatchingGame7.as** file at all. Take a close look at how this movie is put together, along with Figure 3.10, to understand how the game fits into the larger main movie.

Adding a Play Again Button

On the last frame, we want to add another button that enables players to play again.

This is as simple as duplicating the original play button from frame 1. Don't just copy and paste. Instead, create a duplicate of the button in the library. Then, change the text on the button from Play to Play Again.

Your gameover frame should now look like Figure 3.12.

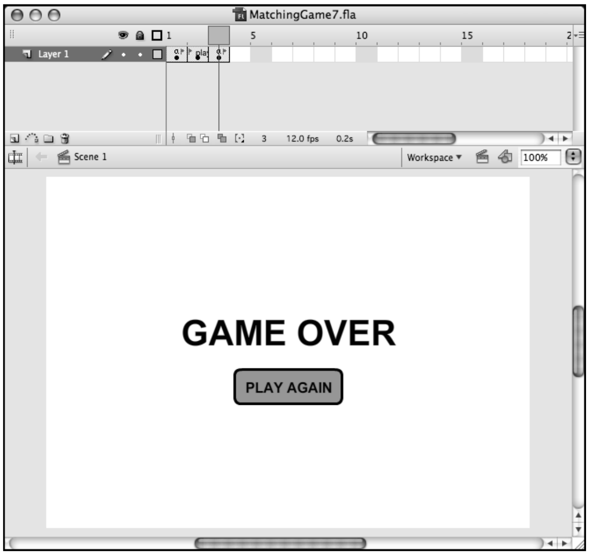


Figure 3.12
The gameover screen now has a Play Again button on it.

After you have added this button to the third frame, name it `playAgainButton` using the Property Inspector so that you can assign a listener to it. The frame script should look like this:

```
playAgainButton.addEventListener(MouseEvent.CLICK,playAgain);

function playAgain(event:MouseEvent) {
    gotoAndStop("playgame");
}
```

Test out **MatchingGame7.fla** and see these buttons in action. You've got a very versatile game framework now, where you can substitute content in the intro and gameover pages, and restart the game without fear of leftover screen elements or variable values. This was quite a problem in ActionScript 1 and 2, but isn't an issue with this sort of framework in ActionScript 3.

Adding Scoring and a Clock

The goal of this chapter is to develop a complete game framework around the basic matching game. Two elements commonly seen in casual games are scoring and timers. Even though the matching game concept doesn't really need them, let's go ahead and add them to the game anyway to make it as full-featured as we can.

Adding Scoring

The first problem is deciding how scoring should work for a game like this. There isn't an obvious answer. However, there should be a positive reward for getting a match, and perhaps a negative response for missing. Because it is almost always the case that a player will miss more than he or she finds matches, a match should be worth far more than a miss. A good starting point is 100 points for a match and –5 points for a miss.

Instead of hard coding these amounts in the game, let's add them to the list of constants at the start of the class:

```
private static const pointsForMatch:int = 100;
private static const pointsForMiss:int = -5;
```

Now, to display the score, we need a text field. Creating a text field is pretty straightforward, as you saw in Chapter 2. We first need to declare a new `TextField` object in the list of class variables:

```
private var gameScoreField:TextField;
```

Then, we need to create that text field and add it as a child:

```
gameScoreField = new TextField();
addChild(gameScoreField);
```

We could also format it and create a nicer-looking text field, as we did in Chapter 2, but we'll leave that part out for now.

The score itself will be a simple integer variable named `gameScore`. We'll declare it at the start of the class:

```
private var gameScore:int;
```

And then we'll set it to zero in the constructor function:

```
gameScore = 0;
```

In addition, it would be a good idea to immediately show the score in the text field:

```
gameScoreField.text = "Score: "+String(gameScore);
```

However, we realize at this point that there are at least several places in the code where we will set the text of `gameScoreField`. The first is in the constructor function. The second will be after the score changes during game play. Instead of copying and pasting the previous line of code in two places, let's move it to a function of its own. Then, we can call the same function from each of the places in the code where we need to update the score:

```
public function showGameScore() {
    gameScoreField.text = "Score: "+String(gameScore);
}
```

We need to change the score in two places in the code. The first is right after we find a match, just before we check to see whether the game is over:

```
gameScore += pointsForMatch;
```

Then, we add an `else` clause to the `if` statement that checks for a match, and subtract points if the match is not found:

```
gameScore += pointsForMiss;
```

Here is the entire section of code so that you can see where these two lines fit in:

```
// compare two cards
if (firstCard.cardface == secondCard.cardface) {
    // remove a match
    removeChild(firstCard);
    removeChild(secondCard);
    // reset selection
    firstCard = null;
    secondCard = null;
    // add points
    gameScore += pointsForMatch;
    showGameScore();
    // check for game over
    cardsLeft -= 2; // 2 less cards
    if (cardsLeft == 0) {
            MovieClip(root).gotoAndStop("gameover");
        }
    } else {
        gameScore += pointsForMiss;
        showGameScore();
    }
```

Notice that we are adding points using the `+=` operation, even if there is a miss. This is because the `pointsForMiss` variable is set to –5. So adding –5 is the same as subtracting 5 points.

We also put in the `showGameScore()` function call after each change to the score. This will make sure the player sees an up-to-date score, as shown in Figure 3.13.

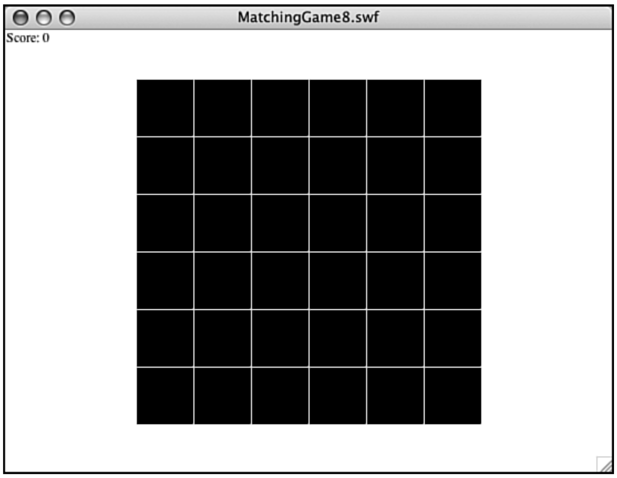


Figure 3.13
The score now appears in the upper left, using the default font and style.

NOTE

In moving from **MatchingGame8.fla** to **MatchingGame9.fla**, you need to do more than just change the filenames. In the movie, you need to change both the name and the class of the **`MatchingGameObject7`** movie clip to **`MatchingGameObject8`**. It would be an easy mistake to only change the name of the movie clip but leave the class pointing to **`MatchingGameObject7`**.

Then, of course, you need to change the name of the ActionScript file to **MatchingGame8.as**, and change the class name and constructor function name, too.

This is true of future versions of the matching game in the rest of this chapter, too.

MatchingGame8.fla and **MatchingGame8.as** include this scoring code. Take a look to see it in action.

Adding a Clock

Adding a clock timer is a little harder than adding a score. For one thing, a clock needs to be updated constantly, as opposed to the score, which only needs to be updated when the user tries a match.

To have a clock, we need to use the `getTimer()` function. This will return the time since the Flash movie started, in milliseconds. This is a special function that requires a special Flash class that we need to import at the start of our program:

```
import flash.utils.getTimer;
```

NOTE

The `getTimer` function measures the number of milliseconds since the Flash movie started. However, it is never useful as a raw time measurement because the player won't ever be starting a game the instant the movie appears on his or her screen. Instead, `getTimer` is useful when you take two measurements and subtract the later one from the earlier one. That is what we will do here, get the time the user pressed Play, and then subtract this from the current time to get the amount of time the game has been played.

Now we will need some new variables. We need one to record the time the game started. Then, we can simply subtract the current time from the start time to get the amount of time the player has been playing the game. We'll also use a variable to store the game time:

```
private var gameStartTime:uint;
private var gameTime:uint;
```

We also need to define a new text field to display the time to the player:

```
private var gameTimeField:TextField;
```

In the constructor function, we add a new text field to display the time. We also move to the right side of the screen so that it won't be on top of the score display:

```
gameTimeField = new TextField();
gameTimeField.x = 450;
addChild(gameTimeField);
```

Before the constructor function is done, we'll want to set the `gameStartTime` variable. We can also set the `gameTime` to zero:

```
gameStartTime = getTimer();
gameTime = 0;
```

Now we need to figure out a way for the game time to update. It will be changing constantly, so we don't want to wait for user action to display the time.

One way to do it is to create a `Timer` object, as in Chapter 2. However, it isn't critical that the clock be updated at regular intervals, only that the clock be updated often enough so that players get an accurate sense of how long they have been playing.

Instead of using a `Timer`, we can just have the `ENTER_FRAME` event trigger a function that updates the clock. In a default Flash movie, this will happen 12 times a second, which is certainly enough:

```
addEventListener(Event.ENTER_FRAME,showTime);
```

All that is left is to make the `showTime` function. It will calculate the current time based on the current value of `getTimer()` and the value of `gameStartTime`. Then, it will put it in the text field for display:

```
public function showTime(event:Event) {
     gameTime = getTimer()-gameStartTime;
     gameTimeField.text = "Time: "+gameTime;
}
```

Figure 3.14 shows the screen with both the score and the current time. However, the time format uses a semicolon and two digits for the seconds. You'll see how to do this next.

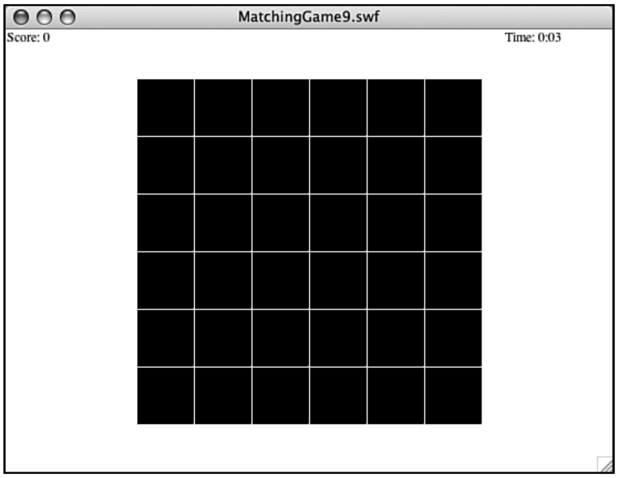


Figure 3.14
The time is now displayed at the upper right.

Displaying Time

The `showTime` function displays the number of milliseconds since the game started. Typical players don't care about milliseconds; they want to see a normal clock, with minutes and seconds displayed as they would on a digital watch.

Let's break this out in another function. Instead of just including the raw `gameTime` in the text field as in the preceding code example, we can call a function to return a nicer output:

```
gameTimeField.text = "Time: "+clockTime(gameTime);
```

The idea is that the old code would show this:

```
Time: 123726
```

The new code will show:

```
Time: 2:03
```

The `clockTime` function will take the time in raw milliseconds and convert it to minutes and whole seconds. In addition, it will format it to use a colon (:) and make sure that a zero is placed correctly when the number of seconds is fewer than ten.

The function will start off by simply dividing the number of milliseconds by 1,000 to get the number of seconds. It will then divide that by 60 to get the number of minutes.

Next, it must subtract the minutes from the seconds. For instance, if there are 123 seconds, that means there are 2 minutes. So, subtract 2*60 from 123 to get 3 seconds left over. 123 is 2 minutes and 3 seconds:

```
public function clockTime(ms:int) {
    var seconds:int = Math.floor(ms/1000);
    var minutes:int = Math.floor(seconds/60);
     seconds -= minutes*60;
```

Now that we have the number of minutes and seconds, we want to make sure that we insert a colon between them, and make sure that the seconds are always two digits.

I use a trick to do this. The `substr` function enables you to grab a set number of characters from a string. So, the number of seconds will be between 0 and 59. Add 100 to that, and you have a number between 100 and 159. Grab the second and third characters from that as a string, and you have a range of 00 to 59. The following line is how it looks in ActionScript:

```
var timeString:String = minutes+":"+String(seconds+100).substr(1,2);
```

Now just return the value.

```
    return timeString;
}
```

The time will now display at the top of the screen in a familiar digital watch format, rather than just as a number of milliseconds.

Displaying Score and Time after the Game Is Over

Before we finish with **MatchingGame9.fla**, let's take the new score and time displays and carry them over to the gameover screen.

This is a little tricky because the gameover screen exists on the main timeline, outside of the game movie clip. To have the main timeline even know what the score and time are, this data needs to be sent from the game to the root level.

Just before we call the `gotoAndStop` command that will advance the movie to the gameover screen, we pass these two values up to `root`:

```
MovieClip(root).gameScore = gameScore;
MovieClip(root).gameTime = clockTime(gameTime);
```

Notice that we pass the score up as a raw value, but we run the time through the handy `clockTime` function so that it is a string with a colon and a two-digit second.

At the root level, we need to define those new variables, which use the same names as the game variables: `gameTime` and `gameScore`. I've added this code to the first frame:

```
var gameScore:int;
var gameTime:String;
```

Then, on the gameover frame, we use these variables to place values in new text fields:

```
showScore.text = "Score: "+String(gameScore);
showTime.text = "Time: "+gameTime;
```

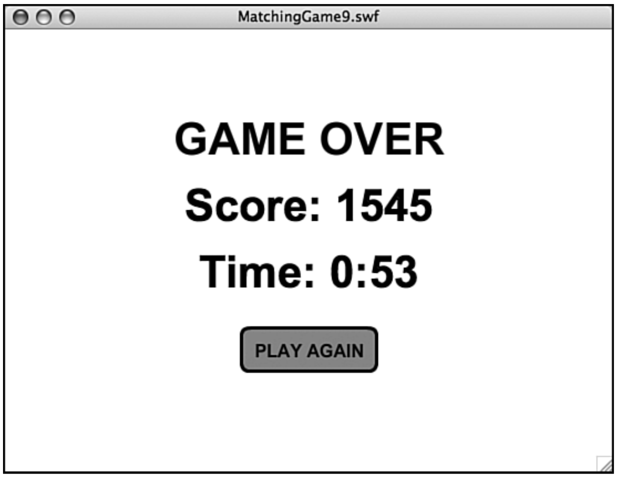
NOTE

To simplify things here, we're including the `"Score: "` and `"Time: "` strings in with the Score and Time fields. But, a more professional way to do it would be to have the words *Score* and *Time* as static text or graphics on the screen, and only the actual score and time in the fields. In that case, encasing the `gameScore` variable inside the `String` function is definitely necessary (because the `.text` property of a text field must be a string). Setting it to just `gameScore` would be trying to set a string to an integer and would cause an error message.

We don't need to use code to create the `showScore` and `showTime` dynamic text fields; we can simply do that on the stage with the Flash editing tools. Figure 3.15 shows what the gameover screen now looks like, when a game is complete.

This completes **MatchingGame9.fla** and **MatchingGameObject9.fla**. We now have a game with an intro and gameover screen. It keeps track of score and time, and displays them when the game is over. It also allows the player to play again.

Next we finish off the game by adding a variety of special effects, such as card flips, limited card-viewing time, and sound effects.



GAME OVER

Score: 1545

Time: 0:53

Figure 3.15
The gameover screen, complete with final score and time.

Adding Game Effects

Gone are the early days of games on the Web, when just the idea of a game in a web page was cool enough to get you attention. Now, you have to work to add quality to your games—little touches such as animation and sound.

Let's spruce up this simple matching game with some special effects. Although they won't change the basic game play, they will make the game seem a lot more interesting to players.

Animated Card Flips

Because we are flipping virtual cards over and back, it makes sense to want to see this flip as an animation. You can do this with a series of frames inside a movie clip, but because you're learning ActionScript here, let's do it with ActionScript.

NOTE

Using a timeline animation rather than an ActionScript one is difficult here because of the nature of the cards. You do not want to animate 18 different cards, just 1. So, you would probably put the card faces inside another movie clip and change the frame of that nested movie clip rather than the main `Card` movie clip. Then the `Card` movie clip can have frames 2 and on, which would be an animated sequence showing a card flip. It is not easy to envision unless you do a lot of Flash animating.

Because this animation affects the cards, and only the cards, it makes sense to put it inside the `Card` class. However, we don't have a `Card` class. We opted way back at the start of this chapter to not use a `Card` class and just allow Flash to assign a default class to it.

Now it is time to create `Card` class. If we make a **Card.as** file, however, it will be used by any `Card` object that is in the folder. We've already got **MatchingGame1.fla** through **MatchingGame9.fla** with `Card` objects in it. So, to make it clear that we only want **MatchingGame10.fla** to use this `Card` class, we'll change the name of the symbol, and the class it references, to `Card10`. Then, we'll create a **Card10.as** ActionScript class file.

This class will enable an animated flip of the card, rather than just changing the card instantly. It will replace all the `gotoAndStop` functions in the main class. Instead, it will tell the card to `startFlip.` It also passes in the frame which the card should show when the flip is over. The `Card10` class will then set up some variables, set up an event listener, and proceed to animate the card over the next ten frames:

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

    public dynamic class Card10 extends MovieClip {
        private var flipStep:uint;
        private var isFlipping:Boolean = false;
        private var flipToFrame:uint;

        // begin the flip, remember which frame to jump to
        public function startFlip(flipToWhichFrame:uint) {
            isFlipping = true;
            flipStep = 10;
            flipToFrame = flipToWhichFrame;
            this.addEventListener(Event.ENTER_FRAME, flip);
        }

        // take 10 steps to flip
        public function flip(event:Event) {
            flipStep--; // next step

            if (flipStep > 5) { // first half of flip
                this.scaleX = .2*(flipStep-6);
            } else { // second half of flip
                this.scaleX = .2*(5-flipStep);
            }

            // when it is the middle of the flip, go to new frame
            if (flipStep == 5) {
                gotoAndStop(flipToFrame);
            }

            // at the end of the flip, stop the animation
            if (flipStep == 0) {
                this.removeEventListener(Event.ENTER_FRAME, flip);
```

```
                    }
                }
            }
        }
```

So, the `flipStep` variable starts at 10 when the `startFlip` function is called. It then is reduced by one each frame thereafter.

NOTE
The `scaleX` property shrinks or expands the width of a movie clip. A value of 1.0 is the default. A value of 2.0 stretches it to twice its width, and a value of .5 makes it half its width.

If `flipStep` is between 6 and 10, the `scaleX` property of the card is set to `.2*(flipStep-6)`, which would be .8, .6, .4, .2, and 0. So, it gets thinner with each step.

Then, when `flipStep` is between 5 and 0, the new formula of `.2*(5-flipStep)` is used. So, it would be 0, .2, .4, .6, .8, and then 1.0; and so, it returns to normal size.

At the fifth step, the card jumps to the new frame. It appears to shrink, goes to nothing, jumps to the new frame, and then grows again.

To accomplish this effect, I had to make one change to how the graphics on the `Card` movie clip were arranged. In all previous versions of the game, the cards had their upper-left corner at the center of the movie clip. For the change to `scaleX` to make it appear that the card was flipping around its center, however, I had to center the card graphics on each frame over the center of the movie clip. Compare the `Card` movie clips in **MachingGame9.fla** and **MatchingGame10.fla** to see the difference. See Figure 3.16 to see how this looks when editing the movie clips.

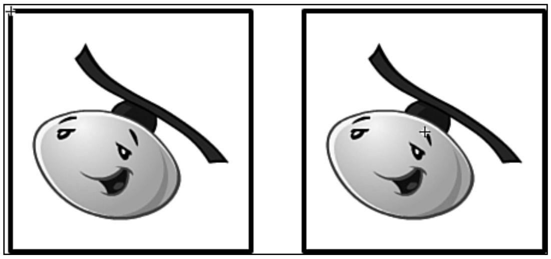


Figure 3.16
The left side shows the registration point of the movie clip at the upper left, as it is in the first nine example movies of this chapter. The right side shows the movie clip centered as it is for the final example.

At the last step, the event listener is removed completely.

The great thing about this class is that it works just as well when the card is being turned back face down, going to frame 1.

Look at **MatchingGameObject10.as** and see where all the `gotoAndStop` calls have been replaced with `startFlip`. By doing this, we are not only creating a flip animation, but we are also giving the Card class more control over itself. Ideally, you might want to give cards complete control over themselves by having the **Card10.as** class more functions, such as those that set the location of the cards at the start of the game.

Limited Card-Viewing Time

Another nice touch to this game is to automatically turn over pairs of mismatched cards after the player has had enough time to look at them. For instance, the player chooses two cards. They don't match, so they remain face up for the player to inspect. After two seconds, however, the cards turn over, even if the player hasn't begun to select another pair.

To accomplish this, we'll use a `Timer`. A `Timer` will make adding this feature relatively easy. To start, we'll need to import the `Timer` class into our main class:

```
import flash.utils.Timer;
```

Next, we create a timer variable at the start of the class:

```
private var flipBackTimer:Timer;
```

Later on in the `clickCard` function, we add some code right after the player has chosen the second card, not made a match, and his or her score has been decreased. This `Timer` code will set up the new timer, which will simply call a function when two seconds have gone by:

```
flipBackTimer = new Timer(2000,1);
flipBackTimer.addEventListener(TimerEvent.TIMER_COMPLETE,returnCards);
flipBackTimer.start();
```

The `TimerEvent.TIMER_COMPLETE` event is triggered when a timer is done. Typically, a `Timer` runs a certain number of times, triggering a `TimerEvent.TIMER` each time. Then, on the last event, it also triggers the `TimerEvent.TIMER_COMPLETE`. Because we only want to trigger a single event at some point in the future, we just set the number of `Timer` events to one, and then look for `TimerEvent.TIMER_COMPLETE`.

When two seconds go by, the `returnCards` function is called. This is a new function that works just like the later part of the old `clickCard` function. It flips both the first and second selections back to the face-down state, and then sets the `firstCard` and `secondCard` values to `null`. It also removed the listener:

```
public function returnCards(event:TimerEvent) {
    firstCard.startFlip(1);
    secondCard.startFlip(1);
    firstCard = null;
    secondCard = null;
    flipBackTimer.removeEventListener(TimerEvent.TIMER_COMPLETE,returnCards);
 }
```

The `returnCards` function duplicates code that was in `clickCard` before, so in **MatchingGameObject10.as** I've replaced this duplicate code in `clickCard` with a simple call to `returnCards`. This way, we only have one spot in our code that returns a pair of cards to the face-down state.

Because `returnCards` demands a single event parameter, we need to pass that parameter into `returnCards` whether we have something to pass or not. So, the call inside `clickCard` simply passes a `null`:

```
returnCards(null);
```

Now if you run the movie and flip two cards, then wait, the cards will flip back on their own.

Because we have a `removeEventListener` command in the `returnCards` function, the listener is removed even if the `returnCards` function is triggered by the player turning over another card. Otherwise, the player would turn over a new card, the first two cards would turn back, and then the event would be triggered after two seconds regardless of the fact that the original two cards are already face down.

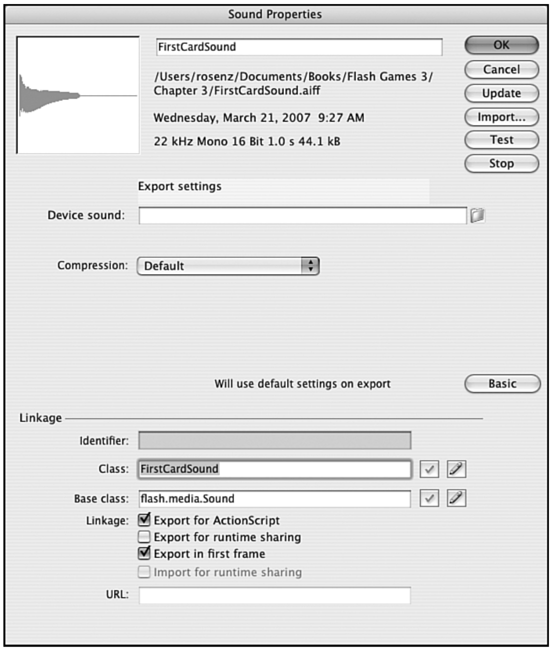
Sound Effects

No game is truly complete without sound. ActionScript 3.0 makes adding sound relatively easy, although there are quite a few steps involved.

The first step is to import your sounds. I've created three sounds and want to bring them each into the library:

FirstCardSound.aiff
MissSound.aiff
MatchSound.aiff

After we have imported them, they need to have properties changed. Name them all after their filenames, but minus the .aiff. Also, check the Export for ActionScript option and give them the same class name as symbol name. Figure 3.17 shows one of the sound's Properties dialog box.



FirstCardSound

/Users/rosenz/Documents/Books/Flash Games 3/Chapter 3/FirstCardSound.aiff

Wednesday, March 21, 2007 9:27 AM

22 kHz Mono 16 Bit 1.0 s 44.1 kB

Export settings

Device sound:

Compression: Default

Will use default settings on export

Linkage

Identifier:

Class: FirstCardSound

Base class: flash.media.Sound

Linkage: [✓] Export for ActionScript
[ ] Export for runtime sharing
[✓] Export in first frame
[ ] Import for runtime sharing

URL:

Figure 3.17
Each sound is a class and can be accessed in ActionScript by its class name.

Next, we set up the main game class to play the sounds at the right time. First, we need to import two new classes so that we can use sound:

```
import flash.media.Sound;
import flash.media.SoundChannel;
```

Then, we create class variables to hold references to these sounds:

```
var theFirstCardSound:FirstCardSound = new FirstCardSound();
var theMissSound:MissSound = new MissSound();
var theMatchSound:MatchSound = new MatchSound();
```

I like to pass all sound playback through a single function. Let's call it `playSound` and add it to the end of the class:

```
public function playSound(soundObject:Object) {
     var channel:SoundChannel = soundObject.play();
}
```

Now, when we want a sound to play, we just call `playSound` with the sound variable we want to use, as follows:

```
playSound(theFirstCardSound);
```

In **MatchingGameObject10.as**, I've added `playSound(theFirstCardSound)` when the first card is clicked, and when a card is clicked while two mismatched cards are already showing. I've added `playSound(theMissSound)` when the second card is turned over and there is no match. I've added `playSound(theMatchSound)` when the second card is turned over and a match is found.

This is all that it takes to add sound effects to the game.

NOTE

You might at this point want to review your publish settings to choose your sound compression settings. Alternatively, you could set the sound compression for each sound individually in its symbol properties. Either way, you probably want to use something pretty low, such as 16Mbps MP3, because these are just simple sound effects.

Modifying the Game

A few more tiny changes before we are done with the game.

First, when we recentered all the cards, it threw off the horizontal and vertical offsets for the card placement. So, that needs to be readjusted:

```
private static const boardOffsetX:Number = 145;
private static const boardOffsetY:Number = 70;
```

How did I come up with those numbers? Well, if you really want to know:

- The stage is 550 pixels across. There are 6 cards, each spaced 52 pixels apart. That's 550–6*52 for the total space remaining on the left and the right. Divide by 2 to get the space to the right. However, the cards are centered at 0,0, so I need to subtract half of the width of a card, or 26. So, `(550-6*52)/2-26=145;`
- Same for the vertical offset. `(400-6*52)/2-26=70;`

Another loose end to consider is the cursor. When users go to click a card, they don't get a special "I can click this" cursor. That is pretty easily changed by just setting the `buttonMode` property of each card as it is created:

```
c.buttonMode = true;
```

Now we have a finger cursor when the user rolls over the cards. This is already the case for the Play and Play Again buttons because those are `Button` symbols.

One last change I made was to increase the frame rate of the movie from the default 12 frames per second to 60. You can do this by choosing Modify, Document to change the main movie document properties.

At 60 frames per second, the flips are much smoother. And with the super-fast ActionScript 3.0 engine, even slow machines can run this game at this high frame rate.

That wraps up the matching game, leaving us with the final version files:

MatchingGame10.fla
MatchingGameObject10.as
Card10.as

4

Brain Games: Memory and Deduction

- Arrays and Data Objects
- Memory Game
- Deduction Game

In the preceding chapter, we looked at a game that had a single setup of a game board, and you played until you cleared the board. However, many games have more than one setup. These games create a situation for the player to deal with, then the player gets to take action, and then the next situation is set up. You can think of these as turn-based games.

In this chapter, we look at two such games: memory and deduction. The first game asks the player to watch and repeat a sequence. Each turn, the sequence gets longer, until the player eventually can't keep up. The second game asks the player to guess a sequence, taking turns and using feedback to try to do better next turn.

The simple setup used in the previous chapter won't work for these games. We need to use arrays and data objects to store information about the game and use these data objects to determine the outcome of each turn the player makes.

Arrays and Data Objects

The games we create in this chapter require that we store information about the game and the player's moves. We use what computer scientists call *data structures* to do this.

Data structures are simply methods for storing groups of information. The simplest data structure is an array. It stores a list of information. ActionScript also has data objects, which store labeled information. In addition, you can nest one inside the other. You can have an array of data objects.

Arrays

An array is a list of values. For instance, if we want to have a list of characters that a player could choose at the start of a game, we could store that list as such:

```
var characterTypes:Array = new Array();
characterTypes = ["Warrior", "Rogue", "Wizard", "Cleric"];
```

We could also use the `push` command to add items to the array. This will produce the same result as the previous code:

```
var characterTypes:Array = new Array();
characterTypes.push("Warrior");
characterTypes.push("Rogue");
characterTypes.push("Wizard");
characterTypes.push("Cleric");
```

In these examples, we are making an array of strings. However, arrays can hold any sort of value, such as numbers or even display objects, such as sprites and movie clips.

NOTE

Not only can arrays store values of any type, but they can mix types. You can have an array like this: `[7, "Hello"]`.

A common use for arrays in games is to store the movie clips and sprites that we create. For instance, in Chapter 3, "Basic Game Framework: A Matching Game," we created a grid of matching cards. For easy access, we could have stored a reference to each `Card` in an array.

Creating the array might have gone something like this, if we want to create ten cards:

```
var cards:Array = new Array();
for(var i:uint=0;i<10;i++) {
    var thisCard:Card = new Card();
    cards.push(thisCard);
}
```

There are many advantages to having your game pieces in an array. For instance, it is easy to loop through them and check each piece for matches or collisions.

NOTE

You can also nest arrays. So, you can have an array of arrays. This is especially useful for grids of game pieces like in Chapter 3. For instance, a tic-tac-toe board could be represented as `[["X","O","O"], ["O","X","O"], ["X","O","X"]]`.

You can add new items to arrays, take items out of arrays, sort them, and search through them. Table 4.1 lists some of the most common array functions.

Table 4.1 Common Array Functions

Function	Example	Description
`push`	`myArray.push("Wizard")`	Adds a value to the end of an array
`pop`	`myArray.pop()`	Removes the last value of an array and returns it
`unshift`	`myArray.unshift("Wizard")`	Adds a value to the beginning of an array
`shift`	`myArray.shift("Wizard")`	Removes the first value in an array and returns it
`splice`	`myArray.splice(7,2, "Wizard","Bard")`	Removes items from a location in the array and inserts new items there
`indexOf`	`myArray.indexOf("Rogue")`	Returns the location of an item, or –1 if it is not found
`sort`	`myArray.sort()`	Sorts an array

Arrays are a common and indispensable data structure used in games. In fact, the rest of the data structures in this section use arrays to turn a single data item into a list if data items.

Data Objects

Arrays are great for storing lists of single values. But what if you want to group some values together? Suppose in an adventure game you want to keep character types, levels, and health together in a group. Say, for instance, a character on the screen would be a "Warrior" at level 15 with a health between 0.0 and 1.0. You could use a data object to store these three pieces of information together.

NOTE

In other programming languages, data objects are the equivalent to associative arrays. Like data objects, associative arrays are a list of items that include a label (a key) and a value. In fact, you can use regular arrays in ActionScript this way, but they aren't as versatile as data objects.

To create a data object, you can define it as a type `Object`. Then, you can add properties to it with dot syntax.

```
var theCharacter:Object = new Object();
theCharacter.charType = "Warrior";
theCharacter.charLevel = 15;
theCharacter.charHealth = 0.8;
```

You could also create this variable the following way:

```
var theCharacter:Object = {charType: "Warrior", charLevel: 15, charHealth: 0.8};
```

Objects are dynamic, meaning that you can add new properties to them whenever you want, and also these properties can be of any variable type. You don't need to declare variables inside an `Object`; you just need to assign a value to them as in the preceding example.

NOTE

Data objects in ActionScript are really not any different from normal objects. In fact, you can even assign a function to a data object. For instance, if you have an object with the properties `firstname` and `lastname`, you could create a function `fullname()` that would return `firstname+" "+lastname`.

Data objects and arrays work very well together. For instance, you could create an array of characters like the one above.

Arrays of Data Objects

We'll be using arrays of data objects to keep track of game elements in almost every game from now on. This will allow us to store the sprites or movie clips themselves, as well as data about them.

For instance, a data object could look like this:

```
var thisCard:Object = new Object();
thisCard.cardobject = new Card();
thisCard.cardface = 7;
thisCard.cardrow = 4;
thisCard.cardcolumn = 2;
```

Now, imagine a whole array of these objects. In the matching game in Chapter 3, we could have put all the cards in objects like this.

Or, imagine a whole set of items on the screen, like in an arcade game. An array of objects would store information about each item, such as speed, behavior, location, and so on.

NOTE
There is another type of object, called a `Dictionary`. Dictionaries can be used just like `Objects`, except you can use any value as a key, such as sprites, movie clips, other objects, and just about anything.

Data structures like arrays and data objects are important in all but the simplest games. Now let's use them in two complete game examples.

Memory Game

SOURCE FILES
http://flashgameu.com
A3GPU04_MemoryGame.zip

A memory game is another simple game played by adults and children alike. It is a rather new game compared to the matching game, which can be played in the absence of technology.

A memory game is where a sequence of images or sounds is presented, and the player tries to repeat the sequence. Usually, the sequence starts with one piece and then adds another with each turn. So, the player first repeats a single item, then two, then three, and so on. For instance: A, then AD, then ADC, then ADCB, then ADCBD, and so on. Eventually, the sequence is long enough that the player makes a mistake and the game is over.

NOTE

Perhaps the most well-known version of the memory game is the 1978 handheld electronic toy Simon. It was created by Ralph Baer, who is considered to be one of the fathers of computer gaming. He created the original Magnavox Odyssey, the first home gaming console. In 2005, he was awarded the National Medal of Technology his role in creating the video game industry.

Preparing the Movie

In ActionScript 3.0 style, we'll create all the game elements in our code. This means starting with a blank main timeline, but not a blank library. The library needs to contain at least the movie clip for the playing pieces, which will be Lights in this case.

We'll have five lights, but all five will be contained in one movie clip. In addition, there needs to two versions of each light: on and off.

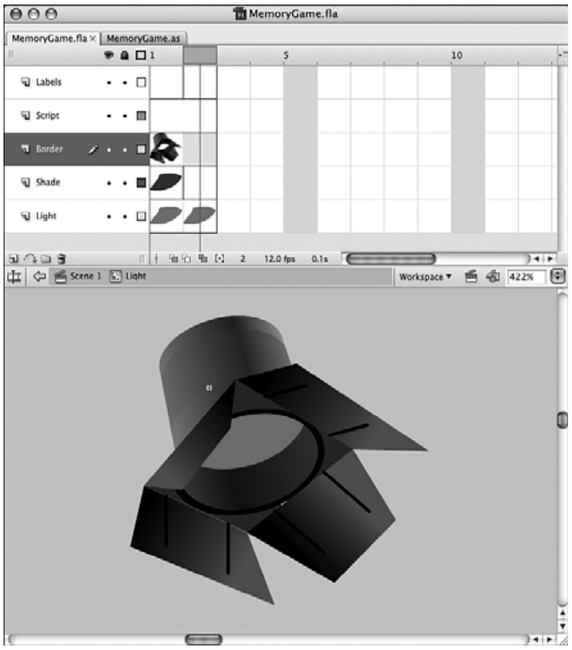


Figure 4.1
The timeline of the `Lights` *movie clip has two frames: off and on. The clip here is shown in Preview mode, which you can access with the pull-down menu on the right side of the timeline.*

The `Lights` movie clip itself, as seen in Figure 4.1, has two frames that both contain another movie clip, `LightColors`. In the first frame of `Lights`, there is a cover over the `LightColors` movie clip that dims its color in the Shade layer. It is a black cover set to 75 percent alpha, which means only 25 percent of the color underneath shows through. So, the first frame is a dim color, which represents the off state of the lights. The second frame is missing the dark cover, so it is the on state.

NOTE

There is no right or wrong way to create game pieces like the memory lights. You could have a movie clip for every light, or every light's state. Or you could have placed all ten variations (five lights times two states) in one ten-frame timeline. Sometimes, it is just a matter of taste. If you are a programmer working with an artist on a game, it might be a matter of accommodating the way the artist wants to create the graphics.

The `LightColors` movie clip contains five frames, each showing a different color. Figure 4.2 shows `LightColors`.

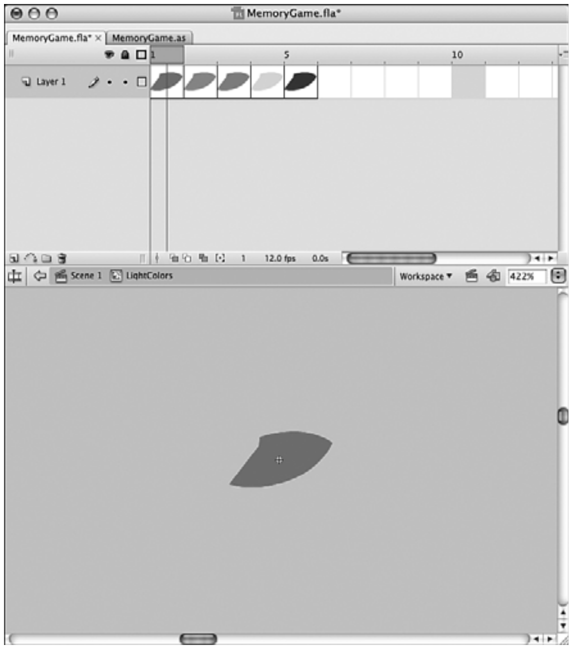


Figure 4.2
The timeline of the `LightColors` *movie clip has one color per frame.*

The `LightColors` movie clip is named lightColors with a lower case *l*. So, to change the color of a Light, we just need to use `lightColors.gotoAndStop` with the frame number.

We'll name the movie **MemoryGame.fla** and the ActionScript file **MemoryGame.as**. So, that means the Document Class needs to be set to MemoryGame in the Property Inspector panel like we did with the Matching Game in Chapter 3.

Programming Strategy

The movie starts with nothing, and then ActionScript creates all the screen elements. Therefore, we need to create the five lights and set each to a color. Then, we need to create two text fields: one to instruct players whether they should be watching the sequence or trying to repeat it, the other letting them know how many lights are in the sequence at the moment.

NOTE

There are plenty of alternatives to using two text fields to display information to the user. For instance, the number of items in the sequence could appear in a circle or box off to one side. The "Watch and listen" and "Repeat" text could instead be symbols that light up like green and red lights on a traffic light. Using text fields is simply a convenient way to not worry about these design elements and focus here on the game logic code.

The `Lights` movie clips will be stored in an array. There are five lights, so that means five elements in the array. This array will make it easy for us to refer to the movie clips when we need to turn them on or off.

We'll also store the sequence in an array. The sequence will start as an empty array, and we'll add one random light to it with each turn.

After a sequence is done playing, we'll duplicate the sequence array. Then, as the player clicks the lights to reproduce the sequence, we'll remove one element from the front of the array with each click. If this element of the sequence matches the click, the player has chosen correctly.

We'll also use `Timer` objects in this game. To play the sequence, a timer will call a function each second to light up a light. Then, a second timer will trigger a function to turn the light off after another half a second passes.

Class Definition

The **MemoryGame.as** will contain the code for this game. Remember to link it to the **MemoryGame.fla** by setting the movie's Document class in the Property Inspector.

To start off the code, we'll declare the package and the class. We need to `import` a few Flash classes. Along with the `flash.display.*` class for showing movie clips, we need the `flash.events.*` class for mouse clicks, the `flash.text.*` class for displaying text, and the `flash.utils.Timer` for using timers. The `flash.media.Sound` and `flash.media.SoundChannel` are needed to play the sounds that accompany the lights. The `flash.net.URLRequest` class is needed to load the sounds from external files:

```
package {
	import flash.display.*;
	import flash.events.*;
	import flash.text.*;
	import flash.utils.Timer;
	import flash.media.Sound;
	import flash.media.SoundChannel;
	import flash.net.URLRequest;
```

NOTE

So, how did I know the names of the classes to import at the start of the code? Simple: I looked them up in the Flash help pages. For instance, to find out what I needed for a text field, I looked up *TextField* and the definition told me right there that it needed `flash.text.*`. Actually, rather than looking for that in the documentation page, I usually skip right to the bottom of the page and look at a code example. The `import` command is easy to find this way.

The class definition includes many variable declarations. The only constant we'll use is the number of lights in the game (in this case, five):

```
public class MemoryGame extends Sprite {
     static const numLights:uint = 5;
```

We'll have three main arrays: one to hold references to the five light movie clips. Then two will hold the sequence of lights. The `playOrder` array will grow with each turn. The `repeatOrder` array will hold a duplicate of the `playOrder` array when the player repeats the sequence. It will shrink as the player clicks lights and comparisons are made with each light in the sequence:

```
private var lights:Array; // list of light objects
private var playOrder:Array; // growing sequence
private var repeatOrder:Array;
```

We need two text fields, one to hold a message to the player at the top of the screen, and one to hold the current sequence length at the bottom of the screen:

```
// text message
private var textMessage:TextField;
private var textScore:TextField;
```

We'll be using two timers in the game. The first will turn on each light in the sequence while it is playing. The second will be used to turn off the lights a half second later:

```
// timers
private var lightTimer:Timer;
private var offTimer:Timer;
```

Other variables needed include `gameMode`, which will store either "play" or "replay" depending on whether the player is watching the sequence, or trying to repeat it. The `currentSelection` variable will hold a reference to the `Lights` movie clips. The `soundList` array will hold references to the five sounds that will play with the lights:

```
var gameMode:String; // play or replay
var currentSelection:MovieClip = null;
var soundList:Array = new Array(); // hold sounds
```

Those are all the variables we need to keep track of. We could have also included constants for the positioning of the text and lights, but we'll hard code them in for the purposes of learning how the code works.

Setting the Text, Lights, and Sounds

The `MemoryGame` constructor function runs as soon as the class is initialized. We'll use it to set up the game screen and load the sounds. Figure 4.3 shows the screen at the very start of the game.

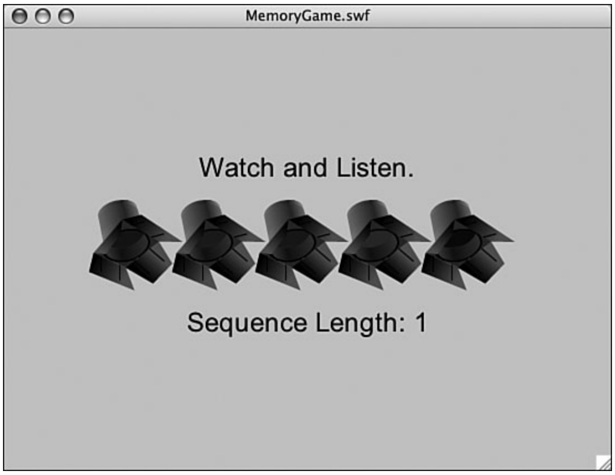


Figure 4.3
The Memory Game screen shows two text fields and five lights.

Adding the Text

Before we set up either text field, we'll create a temporary `TextFormat` object and define how we want the text to look. We'll use this temporary variable when creating both text fields, and then we'll no longer need it. For that reason, we don't need to define the `textFormat` (note the lowercase *t*) variable in the main class, only in this function:

```
public function MemoryGame() {
    // text formating
    var textFormat = new TextFormat();
    textFormat.font = "Arial";
    textFormat.size = 24;
    textFormat.align = "center";
```

The upper text field, to be named `textMessage`, will hold a message to the players to tell them whether they should be watching and listening to the sequence, or whether they need to be clicking lights to repeat the sequence.

We'll place it near the top of the screen. It will be 550 pixels wide, the complete width of the screen. Because `textFormat.align` is set to `"center"`, and the text field is the width of the screen, the text should be centered on the screen.

We also need to set the `selectable` property of the field to `false`; otherwise, the cursor will change to a text selection cursor when the player moves the mouse over the field.

NOTE

Forgetting to set the `selectable` property of a text field to `false` is a common mistake. By default, the `selectable` property is `true`, which means the cursor turns to an editing cursor when the player mouses over it. They can select text; but more important, they cannot easily click objects below the text.

Finally, we set the `defaultTextFormat` to our `textFormat` object to define the font, size and alignment for the field:

```
// create the upper text field
textMessage = new TextField();
textMessage.width = 550;
textMessage.y = 110;
textMessage.selectable = false;
textMessage.defaultTextFormat = textFormat;
addChild(textMessage);
```

The second text field displays the length of the current sequence so that the player can gauge his or her progress. It will be toward the bottom of the screen:

```
// create the lower text field
textScore = new TextField();
textScore.width = 550;
textScore.y = 250;
textMessage.selectable = false;
textScore.defaultTextFormat = textFormat;
addChild(textScore);
```

Loading the Sounds

Next, we'll load the sounds. In Chapter 3, we used sounds that were in the movie's library. ActionScript 3.0 doesn't make this a very versatile way to use sound because each sound in the library needs to be referenced as its own object. Therefore, using five sounds, `"note1"` to `"note5"`, would require five separate objects, and separate lines of code for each.

However, ActionScript 3.0 has a much more robust set of commands for playing external sound files. We'll use those for this game. To set this up, we'll load five sound files, **`"note1.mp3"`** to **`"note5.mp3"`**, into an array of sounds.

NOTE

Flash insists that external sounds be in MP3 format. The great thing about MP3 is that you can really control the size and quality of a file with your audio editing software. So, you can create small, low-quality sounds when it is appropriate to cut down download time, or large, high-quality sounds when it is needed.

```
// load the sounds
soundList = new Array();
for(var i:uint=1;i<=5;i++) {
    var thisSound:Sound = new Sound();
    var req:URLRequest = new URLRequest("note"+i+".mp3");
    thisSound.load(req);
    soundList.push(thisSound);
}
```

NOTE

The "note"+i+".mp3" inside the URLRequest function constructs a string like "note1.mp3". The + symbol concatenates strings and other items into a longer string. So, the result is the concatenation of "note" plus the value of the variable i, plus ".mp3".

Adding the Light Movie Clips

Now that we have text fields and sounds, the last main thing we need to add to the screen is the lights. We'll create five `Light` movie clips and space them so that they are centered. For each `Light` object, we'll send the interior `lightColors` movie clip to a different frame so that each movie clip has a different color.

As well as adding the movie clips to the stage with `addChild`, we'll also add them to the `lights` array for future reference. The `addEventListener` enables the movie clips to react to mouse clicks by calling the `clickLight` function. We'll also set the `buttonMode` property so that the cursor changes to a finger when the user rolls over the light:

```
// make lights
lights = new Array();
for(i=0;i<numLights;i++) {
    var thisLight:Light = new Light();
    thisLight.lightColors.gotoAndStop(i+1); // show proper frame
    thisLight.x = i*75+100; // position
    thisLight.y = 175;
    thisLight.lightNum = i; // remember light number
    lights.push(thisLight); // add to array of lights
    addChild(thisLight); // add to screen
    thisLight.addEventListener(MouseEvent.CLICK,clickLight);
    thisLight.buttonMode = true;
}
```

All of the screen elements have been created. Now it is time to start the game. We'll set the `playOrder` to a fresh empty array, `gameMode`, to `"play"` and then call `nextTurn` to start the first light in the sequence:

```
        // reset sequence, do first turn
        playOrder = new Array();
        gameMode = "play";
        nextTurn();
    }
```

Playing the Sequence

The `nextTurn` function is what kicks off each playback sequence. It adds one random light to the sequence, sets the message text at the top of the screen to "Watch and Listen," and kicks off the `lightTimer` that will display the sequence:

```
    // add one to the sequence and start
    public function nextTurn() {
        // add new light to sequence
        var r:uint = Math.floor(Math.random()*numLights);
        playOrder.push(r);

        // show text
        textMessage.text = "Watch and Listen.";
        textScore.text = "Sequence Length: "+playOrder.length;

        // set up timers to show sequence
        lightTimer = new Timer(1000,playOrder.length+1);
        lightTimer.addEventListener(TimerEvent.TIMER,lightSequence);

        // start timer
        lightTimer.start();
    }
```

So, when a sequence starts playing, the `lightSequence` function gets called each second. The `event` gets passed in as a parameter. The `currentTarget` of this `event` is the equivalent to the `Timer`. The `Timer` object has a property named `currentCount` that returns the number of times the timer has gone off. We'll put that in `playStep`. We can use that to determine which light in the sequence to show.

The function checks the `playStep` to determine whether this is the last time the timer will go off. If so, instead of showing a light, it will start the second half of a turn, where the player needs to repeat the sequence:

```
    // play next in sequence
    public function lightSequence(event:TimerEvent) {
        // where are we in the sequence
        var playStep:uint = event.currentTarget.currentCount-1;
    if (playStep < playOrder.length) { // not last time
        lightOn(playOrder[playStep]);
    } else { // sequence over
        startPlayerRepeat();
    }
}
```

Switching Lights On and Off

When it is time for the player to start repeating the sequence, we'll set the text message to "Repeat." and also the `gameMode` to `"replay"`. Then, we'll make a copy of the `playOrder` list:

```
// start player repetion
public function startPlayerRepeat() {
    currentSelection = null;
    textMessage.text = "Repeat.";
    gameMode = "replay";
    repeatOrder = playOrder.concat();
}
```

NOTE

To make a copy of an array, we use the `concat` function. Although it is meant to create a new array from several arrays, it works just as well to create a new array from only one other array. Why must we do this, instead of just creating a new array and setting it equal to the first one? If we just set one array equal to another, the arrays are literally the same thing. Changing one array changes them both. We want to make a second array that is a copy of the first, so changes to the second array will not affect the first array. The `concat` function lets us do that.

The next two functions turn on and off a `Light`. We'll pass the number of the light into the function. Turning on a light is just a matter of using `gotoAndStop(2)`, which sends the `Light` movie clip to the second frame, the one without the shade covering the color.

NOTE

Instead of using the frames numbers, 1 and 2, we could have also labeled the frames "on" and "off" and used frame label names. This would come in particularly handy in games where there are even more than just these two modes for a movie clip.

We'll also play the sound associated with the light, but using a reference to the sound in the `soundList` array we created.

The only other thing `lightOn` does is to create and start the `offTimer`. This will trigger only one time, 500 milliseconds after the light goes on:

```
// turn on light and set timer to turn it off
public function lightOn(newLight) {
     soundList[newLight].play(); // play sound
     currentSelection = lights[newLight];
     currentSelection.gotoAndStop(2); // turn on light
     offTimer = new Timer(500,1); // remember to turn it off
     offTimer.addEventListener(TimerEvent.TIMER_COMPLETE,lightOff);
     offTimer.start();
}
```

The `lightOff` function then sends the `Light` movie clip back to the first frame. This is where storing a reference to the `Light` movie clip in `currentSelection` comes in handy.

Another thing this function does is to tell the `offTimer` to stop. If the `offTimer` only triggers one time, however, why is this even needed? Well, the `offTimer` will only trigger one time, but `lightOff` could get called twice. This happens if the player repeats the sequence and presses the lights quickly enough that they turn one light off before 500 milliseconds expires. In that case, the `lightOff` gets called once for the mouse click, and then again when the `lightOff` timer goes off. If we issue an `offTimer.stop()` command, however, we can stop this second call to `lightOff`:

```
// turn off light if it is still on
public function lightOff(event:TimerEvent) {
     if (currentSelection != null) {
          currentSelection.gotoAndStop(1);
          currentSelection = null;
          offTimer.stop();
     }
}
```

Accepting and Checking Player Input

The last function needed for the game is the one that is called when the player clicks a `Light` while repeating the pattern.

It starts with a check of the `gameMode` to make sure that the `playMode` is `"replay"`. If not, the player shouldn't be clicking the lights, so the `return` command is used to escape the function.

NOTE

Although `return` is usually used to return a value from a function, it can also be used to terminate a function that isn't supposed to have any returned value at all. In that case, just `return` by itself is all that is needed. However, if the function was supposed to return a value, it needs to be `return` followed by that value.

Assuming that doesn't happen, the `lightOff` function is called to turn off the previous light, if it isn't off already.

Then, a comparison is made. The `repeatOrder` array holds a duplicate of the `playOrder` array. We'll use the `shift` command to pull the first element of the `repeatOrder` array off, and compare it to the `lightNum` property of the light that was clicked.

NOTE

Remember that `shift` pulls an element from the front of an array, whereas `pop` pulls it from the end of the array. If you just want to test the first item in an array, rather than remove it, you can use `myArray[0]`. Similarly, you can use `myArray[myArray.length-1]` to test the last item in an array.

If there is a match, this light is turned on.

The `repeatOrder` will get shorter and shorter as items are removed from the front of the array for comparison. When `repeatOrder.length` reaches zero, the `nextTurn` function is called, and the sequence is added to and played back once again.

If the player has chosen the wrong light, the text message is changed to show the game is over, and the `gameMode` is changed so that no more mouse clicks are accepted:

```
// receive mouse clicks on lights
public function clickLight(event:MouseEvent) {
    // prevent mouse clicks while showing sequence
    if (gameMode != "replay") return;

    // turn off light if it hasn't gone off by itself
    lightOff(null);

    // correct match
    if (event.currentTarget.lightNum == repeatOrder.shift()) {
        lightOn(event.currentTarget.lightNum);

        // check to see if sequence is over
        if (repeatOrder.length == 0) {
            nextTurn();
        }

    // got it wrong
    } else {
        textMessage.text = "Game Over!";
        gameMode = "gameover";
    }
}
```

The gameMode value of "gameover" is not actually used by any other piece of code. Because it is not "repeat", however, clicks won't be accepted by the lights, which is what we want to happen.

All that is left of the code now is the closing brackets for the class and package structures. They come at the end of every AS package file, and the game will not compile without them.

Modifying the Game

In Chapter 3, we started with a game that ran on the main timeline, much like this one. However, at the end of the chapter, we worked to put the game inside a movie clip of its own (leaving the main timeline for introduction and gameover screens).

You could do the same here. Alternatively, you could rename the MemoryGame function to startGame. Then, it would not be triggered at the very start of the movie.

NOTE

If you want to extend this game beyond one frame, you need to change extends Sprite at the start of the class to extends MovieClip. A sprite is a movie clip with a single frame, whereas a movie clip can have more than one frame.

You could put an introduction screen on the first frame of the movie, along with a stop command on the frame and a button to issue the command play so that the movie would continue to the next frame. On that next frame, you could call the startGame function to kick the game off.

Then, instead of just displaying the "Game Over" message when the player misses, you could remove all the lights and the text message with removeChild and jump to a new frame.

Either method, encapsulating the game in a movie clip or waiting to start the game on frame 2, enables you to make a more complete application.

One modification of this game is to start with more than one item in the sequence. You could simply prime the playOrder with two random numbers; the game would then start with a total of three items in the sequence.

Another modification I like that would make it easier to play is to only add new items to the sequence that do not match the last item. For instance, if the first item is 3, the next item can be 1, 2, 4, or 5. Not repeating items one after the other takes a bit of the complexity out of the game.

You could do this with a simple while loop:

```
do {
     var r:uint = Math.floor(Math.random()*numLights);
} while (r == playOrder[playOrder.length-1]);
```

You could also increase the speed at which the sequence plays back. Right now, the lights go on every 1,000 milliseconds. They go off after half of that, 500 milliseconds. So, store 1,000 in a variable (such as `lightDelay`) and then reduce it by 20 milliseconds after each turn. Use its full value for the `lightTimer` and half of its value for `offTimer`.

Of course, the most interesting variations of this game are probably not done with changes to the code, but changes to the graphics. Why do the lights need to be in a straight line? Why do they need to all look the same? Why do they need to be lights at all?

Imagine a game where the lights are in fact songbirds, all different, and all hidden around a forest scene. As they open their beaks and chirp, you need to not only remember which one chirped, but also where it was located.

Deduction Game

SOURCE FILES
http://flashgameu.com
A3GPU04_Deduction.zip

Here we have another classic game. Like the matching game, the game of deduction can be played with a simple set of playing pieces. It can even be played with pencil and paper. However, without a computer, two players are necessary. One player must come up with a somewhat random sequence of colors, while another plays the game to guess the sequence.

NOTE
Deduction is also known under the trademarked name Mastermind as a store-bought physical game. It is the same as a centuries-old game called "Bulls and Cows," which is played with pencil and paper. It is one of the simplest forms of code-breaking games.

The game is usually played with a random sequence of five pegs, each being one of five different colors (for instance: red, green, purple, yellow, blue). The player must make a guess for each of the five spots, although he or she may decline to make a guess for one or more of the spots. So, the player may guess red, red, blue, blue, blank.

When the player guesses, the computer returns the number of pegs correctly placed, and the number of pegs that match the color of a needed peg, although not on the current spot of the peg. So, if the sequence is red, green, blue, yellow, blue, and the player guesses red, red, blue, blue, blank, the result is one correct color and spot, one correct color. It is up to the player to use these two numeric pieces of information to design his or her next guess. A good player can guess the complete sequence usually within ten guesses.

NOTE

Mathematically, it is possible to guess any random sequence with only five guesses. However, this requires some pretty intense calculating. Look up "Mastermind (board game)" at www.wikipedia.org to see details.

Setting Up the Movie

We're going to set this game up is a more robust fashion than the memory game. It will have three frames: an introduction frame, a play frame, and a gameover frame. All three will feature a very simple design so that we can concentrate on the ActionScript.

A background and title will be included across all three frames. You can see it in Figure 4.4.

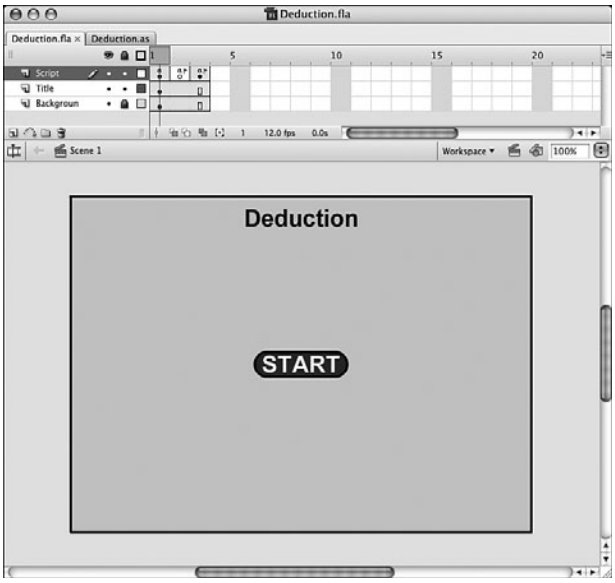


Figure 4.4
Frame 1 features the background and title that run across all the frames, plus a Start button that resides only in frame 1.

Frame 1 has a single button on it. I've created a simple `BasicButton` button display object in the library. It actually has no text on it, but instead the text you see in Figure 4.4 is text laid on top of the button in the frame.

The script in frame 1 stops the movie at the frame and sets the button up to accept a mouse click, which starts the game:

```
stop();
startButton.addEventListener(MouseEvent.CLICK,clickStart);
function clickStart(event:MouseEvent) {
     gotoAndStop("play");
}
```

The second frame, labeled `"play"` in the timeline, will have only a single command. It is a call into our movie class to a function we will create named `startGame`.

```
startGame();
```

The last frame is labeled `"gameover"` and has its own copy of the same button from frame 1. The text over it, however, reads "Play Again." The script in the frame is very similar:

```
playAgainButton.addEventListener(MouseEvent.CLICK,clickPlayAgain);
function clickPlayAgain(event:MouseEvent) {
     gotoAndStop("play");
}
```

In addition to the `BasicButton` library symbol, we need two more. The first is a small button named the `DoneButton`. Figure 4.5 shows this simple button, which does include the text in this case.

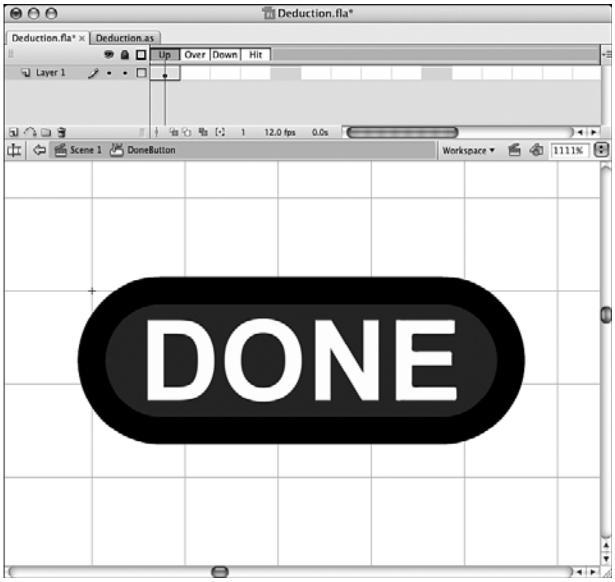


Figure 4.5
The Done button used throughout the game is the same height as the peg holes used in the game.

The main movie clip we need for the game is the `Peg` movie clip. This is actually more than just a peg. It is a series of six frames, the first showing an empty hole, and the other five showing five different colored light bulbs in the hole. Figure 4.6 shows the movie clip.

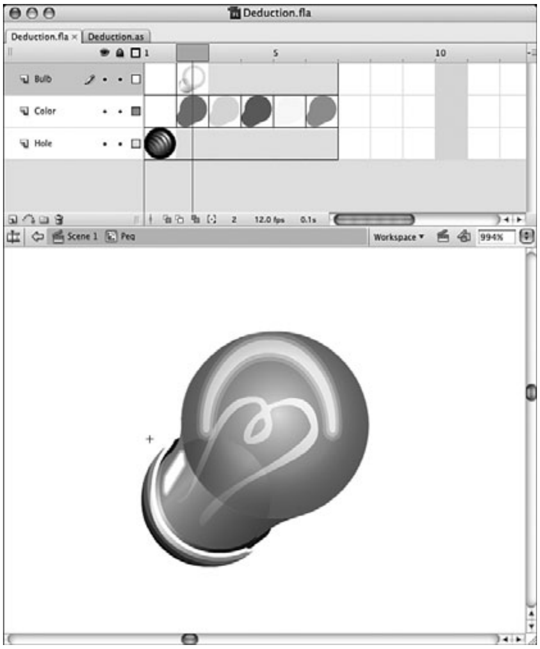


Figure 4.6
The `Peg` *movie clip contains an empty light socket hole and then five frames with the hole filled with different light bulbs.*

Besides the background and title, there is nothing in the main timeline. We'll use ActionScript to create all the game elements. And this time, we'll keep track of every game object created so that we can remove them when the game is over.

Defining the Class

The movie in this example is named **Deduction.fla**, and the ActionScript file is **Deduction.as**. Therefore, the Document class in the Properties panel needs to specify Deduction so that the movie uses the AS file.

The class definition for this game is a little simpler than the class definition for the memory game. For one, we won't be using a timer here, nor will we need any sounds. So, only the `flash.display`, `flash.events`, and `flash.text` classes will be imported—the first to display and control movie clips, the second to react to mouse clicks, and the last to create text fields:

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

For the class declaration, we'll have it extend `MovieClip` rather than `Sprite`. This is because the game will span three frames, not just one. A sprite only uses one frame:

```
     public class Deduction extends MovieClip {
```

For this game, we will use a wide array of constants. To start, we'll define the `numPegs` and `numColors` constants. This way, we can easily change the game to include more than five pegs in the sequence, or more or fewer color options.

We'll also include a set of constants to define where the rows of pegs will be drawn. We'll use a horizontal and vertical offset for all the rows, and the row spacing and peg spacing. This makes it easy to adjust the location of the pegs depending on the size of the pegs and the surrounding elements in the game:

```
// constants
static const numPegs:uint = 5;
static const numColors:uint = 5;
static const maxTries:uint = 10;
static const horizOffset:Number = 30;
static const vertOffset:Number = 60;
static const pegSpacing:Number = 30;
static const rowSpacing:Number = 30;
```

We need two main variables to keep track of progress in the game. The first is an array that holds the solution. It will be a simple array of five numbers (for example, 1, 4, 3, 1, 2). The second variable is `turnNum`, which tracks the number of guesses the player has made:

```
// game play variables
private var solution:Array;
private var turnNum:uint;
```

In this game, we are going to keep very good track of all the display objects we create. There will be a current row of five pegs, stored in `currentRow`. The text to the right of each row of pegs will be `currentText`. The button to the right of the pegs will be `currentButton`. Plus, we'll use the array `allDisplayObjects` to keep track of everything we create:

```
// references to display objects
private var currentRow:Array;
private var currentText:TextField;
private var currentButton:DoneButton;
private var allDisplayObjects:Array;
```

Every class has its constructor function, which shares its name with the class. In this case, however, we won't be using this function at all. This is because the game won't start on the first frame. Instead, it will wait for the player to click the Start button. So we'll include this function, but with no code inside it:

```
public function Deduction() {
}
```

NOTE

It is up to you whether you want to include an empty constructor function. I usually find that I end up needing to do something in the constructor function before I complete a game, so I add it in when starting a new class whether I am sure I need one or not.

Starting a New Game

When a new game starts, the main timeline calls `startGame` to create the sequence of five pegs that the user is seeking. It creates the `solution` array and pushes five random numbers from 1 to 5 into it.

The `turnNum` variable is set to `0`. And then, the workhorse `createPegRow` function is called:

```
// create solution and show the first row of pegs
public function startGame() {
     allDisplayObjects = new Array();
     solution = new Array();
     for(var i:uint=0;i<numPegs;i++) {
          // random, from 1 to 5
          var r:uint = uint(Math.floor(Math.random()*numColors)+1);
          solution.push(r);
     }
     turnNum = 0;
     createPegRow();
}
```

Creating a Row of Pegs

The `createPegRow` function is what does all the work to create the five pegs and the button and text next to them. We'll be calling it each time a turn begins.

It first creates five new copies of the `Peg` object from the library. Each object is placed on the screen according to the values of the constants `pegSpacing`, `rowSpacing`, `horizOffset`, and `vertOffset`. Each object is also set to frame 1, which is the empty hole.

The `addEventListener` command makes each peg react to a mouse click. We also turn on the `buttonMode` property of the pegs so that the cursor changes over them.

The `pegNum` property is added to each peg as it is created. This helps to identify which peg has been clicked.

After the peg is added to the screen with `addChild`, it is also added to `allDisplayObjects`. Then, it is added to `currentRow`, but not by itself. Instead, it is added to `currentRow` as a small object with the properties `peg` and `color`. The first is a reference to the movie clip. The second is a number defining the color, or lack thereof, of the peg in the hole:

```
// create a row of pegs, plus the DONE button and text field
public function createPegRow() {

     // create pegs and make them buttons
     currentRow = new Array();
     for(var i:uint=0;i<numPegs;i++) {
          var newPeg:Peg = new Peg();
          newPeg.x = i*pegSpacing+horizOffset;
          newPeg.y = turnNum*rowSpacing+vertOffset;
          newPeg.gotoAndStop(1);
          newPeg.addEventListener(MouseEvent.CLICK,clickPeg);
          newPeg.buttonMode = true;
          newPeg.pegNum = i;
          addChild(newPeg);
          allDisplayObjects.push(newPeg);

          // record pegs as array of objects
          currentRow.push({peg: newPeg, color: 0});
     }
```

After the five pegs have been created, a copy of the `DoneButton` movie clip is added to the right. But first, a check is made to see whether the `currentButton` already exists. It won't exist the very first time a row of pegs is made, so the button is created, and added to the stage. It also gets an event listener and is added to `allDisplayObjects`.

The horizontal location of the button is determined by constants. It should be one more `pegSpacing` to the right of the last peg in the row. We only need to set the `x` coordinate of the button when it is first created because we are simply moving it further down the screen with every new row of pegs added. So, the `x` position is only set this one time, but the `y` position is set each and every time the `createPegRow` function is called:

```
     // only create the DONE button if we haven't already
     if (currentButton == null) {
          currentButton = new DoneButton();
          currentButton.x = numPegs*pegSpacing+horizOffset+pegSpacing;
          currentButton.addEventListener(MouseEvent.CLICK,clickDone);
          addChild(currentButton);
          allDisplayObjects.push(currentButton);
     }
     // position DONE button with row
     currentButton.y = turnNum*rowSpacing+vertOffset;
```

Adding the Text Field

After the button comes the text field. This is positioned one more `pegSpacing` plus the width of the `currentButton` to the right. We won't do any special formatting here, to keep things simple.

Unlike the button, a new text field will be added each time we create a row of pegs. To solve the puzzle, players must be able to look back at all of their guesses and check the results.

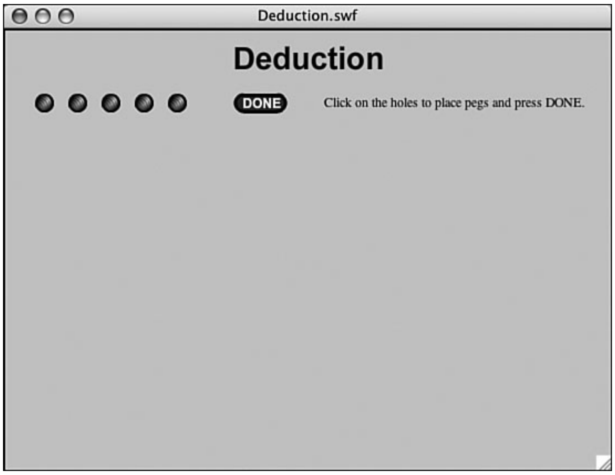
NOTE

The `currentButton` is defined as a `DoneButton` at the start of the class. However, it isn't assigned a value. So, when this function first goes to use it, the value is `null`. Most objects will be set to `null` when they are first created. However, numbers will be set to zero and can never be set to `null`.

The text field starts out with the instructions "Click on the holes to place pegs and click DONE." However, it will later hold the results of each guess at the end of each turn:

```
      // create text message next to pegs and button
      currentText = new TextField();
      currentText.x = numPegs*pegSpacing+horizOffset+pegSpacing*2+currentButton.width;
      currentText.y = turnNum*rowSpacing+vertOffset;
      currentText.width = 300;
      currentText.text = "Click on the holes to place pegs and click DONE.";
      addChild(currentText);
      allDisplayObjects.push(currentText);
  }
```

Take a look at Figure 4.7 to see what the screen looks like when the first `createPegRow` is called. You will see the five pegs, followed by the button, and then the text field.



Click on the holes to place pegs and press DONE.

Figure 4.7
When the game starts, the first row of peg holes is created, and then the player must make their first guess.

Checking Player Guesses

When the player clicks a peg hole, it cycles through the colored pegs, and back to an empty hole if the player clicks it enough times.

We'll determine which `Peg` the player has clicked by looking at the `event.currentTarget.pegNum`. That will give us an index to look up in the `currentRow` array. From there, we can get the `color` and `Peg` properties.

NOTE

The colors are numbered one through five, with zero representing the absence of a peg. However, frames in movie clips are numbered starting with one. So frame 1 is the empty hole, and frames 2 through 6 are the colors. Keep that in mind when looking at the code and the +1 in the `gotoAndStop` statement.

The color of the `Peg` is stored in `currentColor`, a temporary variable used only in this function. If this is less than the number of colors available, the `Peg` simply moves to show the next color. If the last color is already showing, the `Peg` cycles back to show the empty hole, which is the first frame of the page:

```
// player clicks a peg
public function clickPeg(event:MouseEvent) {
     // figure out which peg and get color
     var thisPeg:Object = currentRow[event.currentTarget.pegNum];
     var currentColor:uint = thisPeg.color;

     // advance color of peg by one, loop back from 5 to 0
     if (currentColor < numColors) {
          thisPeg.color = currentColor+1
     } else {
          thisPeg.color = 0;
     }

     // show peg, or abscence of
     thisPeg.peg.gotoAndStop(thisPeg.color+1);
}
```

To play the game, the player will most likely move the cursor over each of the five holes in the current row and click the required number of times to get the hole to show the color of peg he or she wants to guess. After doing this for all five holes, the player then moves on by clicking the Done button.

Evaluating Player Moves

When the player clicks the Done button, the `clickDone` function is called. This function then hands off to the `calculateProgress` function:

```
// player clicks DONE button
public function clickDone(event:MouseEvent) {
     calculateProgress();
}
```

The `calculateProgress` function does all the work to determine how the player's guess matches the `solution`.

The two local variables, `numCorrectSpot` and `numCorrectColor`, are the main results we are looking to calculate. Figuring out the `numCorrectSpot` is actually very easy: Just loop through each `Peg` and determine whether the color selected by the player matches the `solution`. If it does, add one to `numCorrectSpot`.

However, calculating `numCorrectColor` is much more complex. First, you want to ignore any pegs that the user got right. So, only concentrate on the incorrect pegs. Look at the colors the player selected, and determine which ones could have fit in elsewhere.

A clever way to so this is to keep track of each color the player used in these incorrect pegs. Also, keep track of the colors needed in the incorrect pegs. We'll do that with the arrays `solutionColorList` and `currentColorList`.

Each of the items in these arrays will be a sum of each color found. For instance, if two reds (color 0), one green (color 1), and one blue (color 2) are found, the resulting array would be `[2,1,1,0,0]. 2+1+1=4`. Since there are five pegs, and the sum is four, we must have one peg correct, and only four peg holes need to be solved.

So, if `[2,1,1,0,0]` represents the colors used by the player in the wrong pegs, and `[1,0,1,2,0]` represents the colors in those locations that were needed, we can determine the number of correct colors used in the wrong locations by simply taking the minimum number from both arrays: `[1,0,1,0,0]`.

Let's look at that color by color. In the first array, `[2,1,1,0,0]`, the player places two reds. But the second array, `[1,0,1,2,0]`, shows that only one red is needed. So, the minimum number between two and one is one. Only one red is out of place. The player also placed one green. But, the second array shows zero greens are needed. So, the smaller number is zero. The player picked one blue, and one blue is needed. That is another one in the array. The player picked zero yellows, but two were needed. The player picked zero purples, and zero are needed. That is another zero in the array. The minimum is zero. So, 1+0+1+0+0 = 2. Two colors are misplaced. See Table 4.2 to see another view of this calculation.

Table 4.2 Calculating Misplaced Pegs

Color	User Chose	Solution Uses	Number of Pegs Misplaced
Red	2	1	1
Green	1	0	0
Blue	1	1	1
Yellow	0	2	0
Purple	0	0	0
Total misplaced			2

In addition to calculating the total pegs correct and colors misplaced, we'll also take this opportunity to turn off the pegs as buttons by using `removeEventListener` and setting `buttonMode` to `false`:

```
// calculate results
public function calculateProgress() {
     var numCorrectSpot:uint = 0;
     var numCorrectColor:uint = 0;
     var solutionColorList:Array = new Array(0,0,0,0,0);
     var currentColorList:Array = new Array(0,0,0,0,0);

     // loop through pegs
     for(var i:uint=0;i<numPegs;i++) {
          // does this peg match?
          if (currentRow[i].color == solution[i]) {
               numCorrectSpot++;
          } else {
               // no match, but record colors for next test
               solutionColorList[solution[i]-1]++;
               currentColorList[currentRow[i].color-1]++;
          }
          // turn off peg as a button
          currentRow[i].peg.removeEventListener(MouseEvent.CLICK,clickPeg);
          currentRow[i].peg.buttonMode = false;
     }

     // get the number of correct colors in right place
     for(i=0;i<numColors;i++) {
          numCorrectColor += Math.min(solutionColorList[i],currentColorList[i]);
     }
```

Now that we know the results of the tests, we can display them in the text field that previously held the instructions:

```
     // report results
     currentText.text = "Correct Spot: "+numCorrectSpot+", Correct Color:
"+numCorrectColor;
```

Next, we want to advance the `turnNum` and check to see whether the player has found the solution. If so, we'll pass off control to the `gameOver` function. In addition, if the player has exceeded the maximum number of tries, we'll pass control to the `gameLost` function.

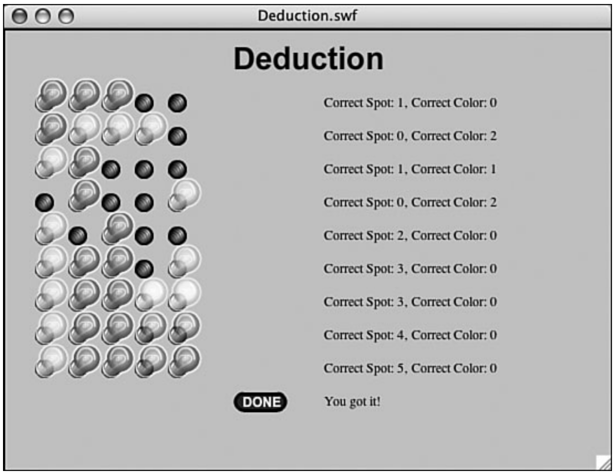
If the game isn't over, we'll go on to the next turn by calling `createPegRow`:

```
     turnNum++;

     if (numCorrectSpot == numPegs) {
          gameOver();
```

```
        } else {
            if (turnNum == maxTries) {
                gameLost();
            } else {
                createPegRow();
            }
        }
    }
```

Ending the Game

If the player has found the solution, we want to tell them the game is over. We'll go ahead and create one more row in the game. This time, there is no need to show the pegs because the previous line that the player completed now holds the matching solution. Figure 4.8 shows what the screen might look like.



Correct Spot: 1, Correct Color: 0
Correct Spot: 0, Correct Color: 2
Correct Spot: 1, Correct Color: 1
Correct Spot: 0, Correct Color: 2
Correct Spot: 2, Correct Color: 0
Correct Spot: 3, Correct Color: 0
Correct Spot: 3, Correct Color: 0
Correct Spot: 4, Correct Color: 0
Correct Spot: 5, Correct Color: 0
You got it!

Figure 4.8
The game ends when the player finds the solution.

Showing the Player Has Won

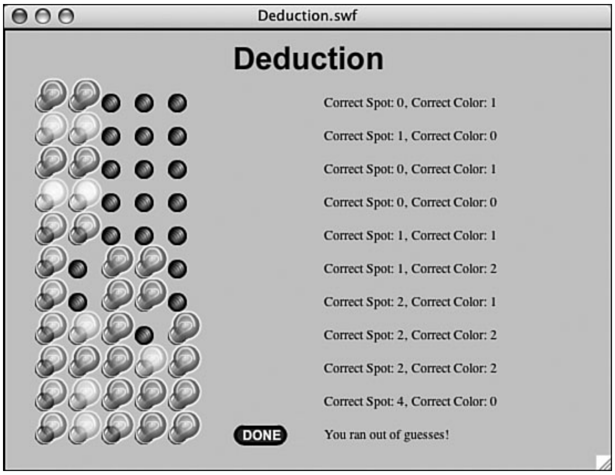
The new row only needs to contain the button and a new text field. The button will be rewired to trigger the `clearGame` function. The next text field will display "You Got It!" We need to add this to `allDisplayObjects` just like any other thing we create in the game:

```
// player found the solution
public function gameOver() {
    // change the button
    currentButton.y = turnNum*rowSpacing+vertOffset;
    currentButton.removeEventListener(MouseEvent.CLICK,clickDone);
    currentButton.addEventListener(MouseEvent.CLICK,clearGame);

    // create text message next to pegs and button
```

```
        currentText = new TextField();
        currentText.x = numPegs*pegSpacing+horizOffset+pegSpacing*2+currentButton.width;
        currentText.y = turnNum*rowSpacing+vertOffset;
        currentText.width = 300;
        currentText.text = "You got it!";
        addChild(currentText);
        allDisplayObjects.push(currentText);
}
```

Showing the Player Has Lost

The `gameLost` function is similar to the `gameOver` function. The main difference is that it needs to create a final row of pegs to reveal the solution to the perplexed player. Figure 4.9 shows what the screen might look like at this point.



Deduction.swf

Deduction

Correct Spot: 0, Correct Color: 1
Correct Spot: 1, Correct Color: 0
Correct Spot: 0, Correct Color: 1
Correct Spot: 0, Correct Color: 0
Correct Spot: 1, Correct Color: 1
Correct Spot: 1, Correct Color: 2
Correct Spot: 2, Correct Color: 1
Correct Spot: 2, Correct Color: 2
Correct Spot: 2, Correct Color: 2
Correct Spot: 4, Correct Color: 0
You ran out of guesses!

Figure 4.9
This player has run out of guesses.

These new pegs don't need to be wired up as buttons. However, they do need to be set to display the correct peg color.

In addition, the Done button is changed to call the `clearGame` function, the same as in `gameOver`. A new text field is created, but this time it displays "You ran out of guesses":

```
// player ran out of turns
public function gameLost() {
        // change the button
        currentButton.y = turnNum*rowSpacing+vertOffset;
        currentButton.removeEventListener(MouseEvent.CLICK,clickDone);
        currentButton.addEventListener(MouseEvent.CLICK,clearGame);

        // create text message next to pegs and button
        currentText = new TextField();
        currentText.x = numPegs*pegSpacing+horizOffset+pegSpacing*2+currentButton.width;
        currentText.y = turnNum*rowSpacing+vertOffset;
        currentText.width = 300;
        currentText.text = "You ran out of guesses!";
        addChild(currentText);
        allDisplayObjects.push(currentText);

        // create final row of pegs to show answer
        currentRow = new Array();
        for(var i:uint=0;i<numPegs;i++) {
            var newPeg:Peg = new Peg();
            newPeg.x = i*pegSpacing+horizOffset;
            newPeg.y = turnNum*rowSpacing+vertOffset;
            newPeg.gotoAndStop(solution[i]+1);
            addChild(newPeg);
            allDisplayObjects.push(newPeg);
        }

    }
```

When the game ends, the Done button remains on the screen, in the final row. Clicking it takes the player to the gameover screen on the main timeline where the player can elect to play again. Before we do that, however, we need to clear the stage of all game elements.

Clearing Game Elements

Removing display objects is a multistep process made a lot easier by our `allDisplayObjects` array. Every single display object we created, whether it was a movie clip, button, or text field, was added to this array. Now we can remove them by just looping through the array and using `removeChild` to take the object off the screen:

```
    // remove all to go to gameover screen
    public function clearGame(event:MouseEvent) {
        // remove all display objects
        for(var i in allDisplayObjects) {
            removeChild(allDisplayObjects[i]);
        }
```

Even with the objects off of the stage, they still exist, waiting for an `addChild` command to place them back on the screen. To really get rid of them, we need to remove all references to these objects. We have a number of places where we refer to display objects, including the `allDisplayObjects` array. If we set that to `null`, and then set the `currentText`, `currentButton`, and `currentRow` variables all to `null`, none of the variables refer to any of our display objects any more. The objects are then deleted.

NOTE

Actually, when you remove all references to a display object, it becomes eligible for "trash collection." This simply means that Flash can remove these objects from memory at any time. Because there are no references to these objects any more, however, and no way to refer to one of the objects even if you want to, you can consider them deleted. Flash won't waste time deleting them right away, however, only when it has a few spare processor cycles to do so.

```
        // set all references of display objects to null
        allDisplayObjects = null;
        currentText = null;
        currentButton = null;
        currentRow = null;
```

Finally, the `clearGame` function tells the main timeline to go to the `"gameover"` frame. This is where there is a Play Again button waiting for the player. And with all the display objects removed, the game can really start again, with fresh new display objects:

```
        // tell main timeline to move on
        MovieClip(root).gotoAndStop("gameover");
    }
```

The `clearGame` function is the critical one used for building a game on the main timeline that can be cleared and restarted. Compared to the way the matching game from Chapter 3 works, they seem to have identical results. You can be assured that the second time the player starts the game, it will behave as a fresh new game, just like the first time.

However, the approach used in Chapter 3 is a little easier to implement because all the display objects are instantly and easily removed when the game movie clip disappears in the game over frame.

Modifying the Game

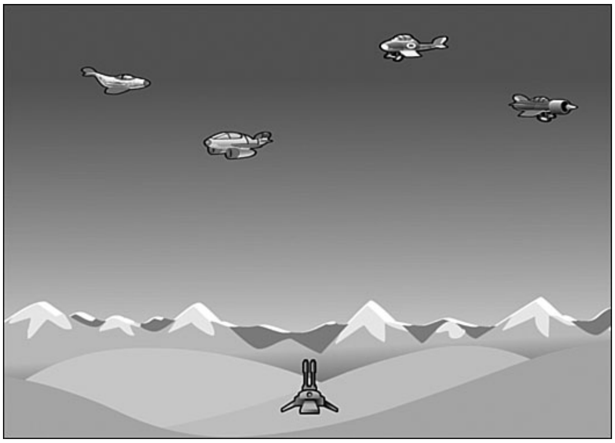
Like the memory game, some of the best variations on deduction are graphics based. You can use almost any sort of object for the pegs, and even create a story to support the graphics. For instance, you could be trying to crack open a safe, or unlock a door in an adventure game.

Our use of constants makes it easy for this game to support more guesses, fewer or more pegs or colors, or any combination. If you want to use more guesses, you'll probably need a longer stage size, or shorter row spacing.

To complete this game, I'd first use a `TextFormat` object to format the message text. Then, I'd add instructions to the introduction screen. A Restart button on the play frame would allow the player to start over at any time. It could simply call `clearGame` to remove all screen elements and go to the game over frame.

To make this game something more like the physical Mastermind game, you'd want to replace the message text with black and white pegs. One would signify a correct peg; the other would signify a correct color in the wrong spot. So black, black, white would mean two correct pegs and one correct color in the wrong spot.

This page intentionally left blank



5

Game Animation: Shooting and Bouncing Games

- Game Animation
- Air Raid
- Paddle Ball

So, far, we have only built games where the game pieces stay in one place. They change and accept player input, but they don't move.

In this chapter, we work with animated game pieces. Some will be controlled by the player, and others will have a life of their own.

After looking at some animation examples, we'll build two games. The first is Air Raid, a simple game in which you control an anti-aircraft gun and try to shoot down planes flying overhead. The second game is Paddle Ball, in which you control a paddle and direct a ball up into a field of blocks that disappear when the ball hits them.

Game Animation

SOURCE FILES
http://flashgameu.com
A3GPU05_Animation.zip

In Chapter 2, "ActionScript Game Elements," we looked at two main types of animation: frame based and time based. We use only time-based animation in this chapter because it is the most reliable and provides the best-looking results.

Time-Based Animation

The basic idea of time-based animation is to move objects at a consistent rate, no matter how good or bad the performance of the Flash player.

A single movement, which we will call a step, takes place every frame. So, a frame rate of 12 frames per second should mean 12 steps per frame. Even though the animation steps occur in every frame, this will not be frame-based animation because we will determine the size of each step based on the time.

For each step, we'll calculate the time since the last step was taken. Then, we will move the game pieces according to this time difference.

So, if the first step takes 84 milliseconds, and the second step takes 90 milliseconds, we'll be moving things slightly farther in the second step than in the first step.

Figure 5.1 shows a diagram of what three frames of movement using time-based animation might look like.

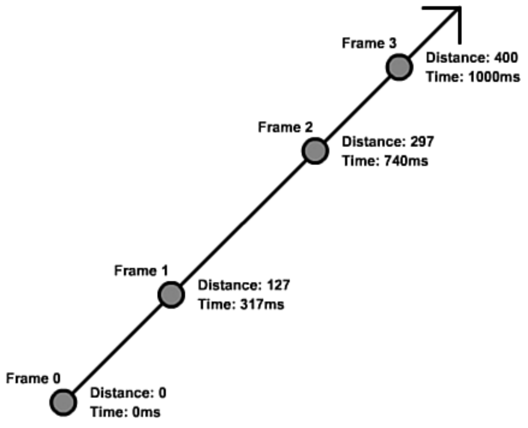


Figure 5.1
The object travels 400 pixels in 1 second regardless of the frame rate.

The object in Figure 5.1 is supposed to move 400 pixels in 1 second. The movie is set to a very slow four frames per second. To complicate things, the computer is being unresponsive, perhaps dealing with other applications or network activity. It is unable to deliver an even frame rate, or even to generate four frames per second.

NOTE

When developing with time-based animation it is a good idea to frequently change the frame rate of your movie while testing. I usually switch back and forth between 12 and 60 frames per second. My goal is to have something that looks great at 60 frames per second, but plays just as well at 12 frames per second.

If I accidentally tie some game element to the frame rate rather than the time, I can quickly see there is a huge difference between game play in the two frame rates.

When the first frame passes, and an `ENTER_FRAME` event triggers the animation function on our code, 317 milliseconds have passed. At 4 frames per second, only 250 milliseconds should have passed. So, the frame rate is already lagging.

But by using the time of 317 milliseconds, we can calculate that the object should have moved the distance of 127 pixels. That's 400 pixels per second, times .317 seconds. So, at the first frame, the object is exactly where it needs to be.

The second frame takes even longer, an additional 423 milliseconds. This is a total of 740 milliseconds, which places the object at 297 pixels out.

Then, in the final frame of the example, an additional 260 milliseconds goes by. This puts the time at exactly 1,000 milliseconds. The distance is 400. So, after 1 second, the object has moved 400 pixels, despite the fact that the movie delivered an inconsistent frame rate and failed to generate 4 frames in the second.

If we had simply made the object move 100 pixels every frame, it would now be at 300 pixels out, having gotten only 3 frames to move. This could be different on another computer or at another time on the same computer, when performance was good enough to deliver 4 frames per second.

Coding Time-Based Animation

The trick to coding time-based animation is to keep track of the time. By looking at the function `getTimer`, you can get the number of milliseconds since the movie started. The actual value of `getTimer` isn't important. It is the difference in time between frames that matters.

For instance, it may take 567 milliseconds for your movie to initialize and place items on the screen. So, the first frame happens at 567 and the second frame at 629. The difference is 62 milliseconds, which is what we need to know to determine how far an object has moved between the frames.

The movie **AnimationTest.fla** contains a simple circle movie clip to demonstrate time-based animation. The movie uses **AnimationTest.as** as its main script and **AnimatedObject.as** as the class for the movie clip.

The `AnimatedObject` class has an constructor function that accepts parameters. This means that when you create a new `AnimatedObject`, you must pass parameters in, like this:

```
var myAnimatedObject:AnimatedObject = new AnimatedObject(100,150,5,-8);
```

The four parameters represent the horizontal and vertical location of the movie clip, plus the horizontal and vertical speed of the movie clip.

Here is the class declaration, variable declarations, plus the `AnimatedObject` function. You can see the four parameters, defined very simply as `x, y, dx, dy`:

```
package {
    import flash.display.*;
    import flash.events.*;
    import flash.utils.getTimer;

    public class AnimatedObject extends MovieClip {
        private var speedX, speedY:Number; // current speed, pixels per second
        private var lastTime:int; // remember the last frame's time

        public function AnimatedObject(x,y,dx,dy) {
            // set location and speed
            this.x = x;
            this.y = y;
            speedX = dx;
            speedY = dy;
            lastTime = getTimer();
            // move each frame
            addEventListener(Event.ENTER_FRAME, moveObject);
        }
```

NOTE

Using `dx` and `dy` to store "difference in x" and "difference in y" is a pretty common practice. In this chapter and the following ones, we use these two variables names often.

The function takes the four parameters and applies them. The first two are used to set the location of the movie clip. The other two are stored in `speedX` and `speedY`.

Then, the variable `lastTime` is initialized with the current value of `getTimer()`. Finally, `addEventListener` will allow the function `moveObject` to run every frame.

The `moveObject` function first calculates the time passed, and then adds that to the `lastTime`. The value of `timePassed` is then used in calculating the change in location.

NOTE

By adding `timePassed` to `lastTime`, you ensure that no time is lost in the animation. If you instead set `lastTime` to `getTimer()` with every animation step, you might lose small slices of time between the calculation of `timePassed` and the setting of `lastTime`.

Because `timePassed` is in thousandths of a second (milliseconds), we divide by 1,000 to get the correct amount to multiply by `speedX` and `speedY`. For instance, if `timePassed` is 100, that is the same as 100/1000 or .1 seconds. If `speedX` is 23, the object moves 23*.1 or 2.3 pixels to the right:

```
        // move according to speed
        public function moveObject(event:Event) {
            // get time passed
            var timePassed:int = getTimer() - lastTime;
            lastTime += timePassed;

            // update position according to speed and time
            this.x += speedX*timePassed/1000;
            this.y += speedY*timePassed/1000;
        }
    }
}
```

A simple way to test this `AnimatedObject` class is with a main movie class like this:

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

```
     public class AnimationTest extends MovieClip {

          public function AnimationTest() {
               var a:AnimatedObject = new AnimatedObject(100,150,5,-8);
               addChild(a);
          }
     }
}
```

This would create a new movie clip at 100,150 that is moving at a speed of 5 horizontally and –8 vertically. So, the `AnimatedObject` class has essentially allowed us to add a moving object to the stage with only two lines of code.

A better test of the `AnimatedObject` class is to add multiple objects and have them all move around in random directions. Here is a version of the main movie class that does just that:

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

     public class AnimationTest extends MovieClip {

          public function AnimationTest() {
               // create 50 objects at random locations with random speeds
               for(var i:uint=0;i<50;i++) {
                    var a:AnimatedObject =
                                   new AnimatedObject(Math.random()*550, Math.ran-
                                   dom()*400, getRandomSpeed(), getRandomSpeed());
                    addChild(a);
               }
          }

          // get a speed from 70-100, positive or negative
          public function getRandomSpeed() {
               var speed:Number = Math.random()*70+30;
               if (Math.random() > .5) speed *= -1;
               return speed;
          }
     }
}
```

In this version of the class, we create a new `AnimatedObject` with a random location and a random speed. The random location is made by the simple use of `Math.random`. For a random speed, however, I used a separate function that returns a value between 70 and 100, positive or negative. This is to prevent having objects that are moving close to 0 speed in a direction.

Figure 5.2 shows this movie when it first runs. The objects are scattered on the screen.

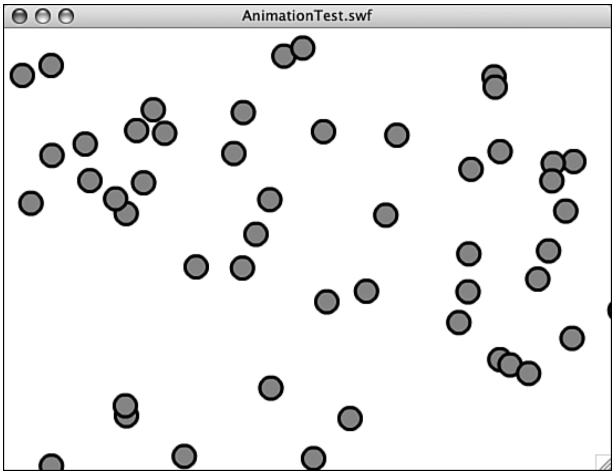


Figure 5.2
The AnimationTest movie places 50 random objects on the stage.

You can play with this class a bit to create some interesting effects. For instance, if you have all the objects starting at the same location, you get an explosion effect.

You can also adjust both the number of objects created and the frame rate of the movie to see how well your computer handles a heavy load of animation.

Now let's use this technique in a game that has three different types of animated objects.

Air Raid

SOURCE FILES
http://flashgameu.com
A3GPU05_AirRaid.zip

Air Raid is similar to many early arcade games. Most of these were naval themed, where you played a sub commander shooting up at ships on the surface. The earliest was probably Sea Wolf, which featured a fake periscope that you looked through and used to aim. It was actually a video game version of competing electronic games called Periscope, Sea Raider, and Sea Devil.

NOTE
Naval torpedo games were probably easier to make in the early days of computer games because ships and torpedoes moved slowly compared to planes and anti-aircraft fire.

In our game, the player moves an anti-aircraft gun along the bottom of the screen with the keyboard arrows. The player fires straight up at passing planes and tries to hit as many as possible with a limited amount of ammo.

Movie Setup and Approach

This game is a perfect opportunity to create a game that uses multiple classes. We've got essentially three different types of objects: airplanes, the turret, and bullets. By creating a single class for each, we can build the game step by step, and then specialize the code for each.

We need three movie clips to go with the three classes. The `AAGun` and `Bullet` movie clips will be one frame each. However, the `Airplane` movie clip will be several frames, each with a different drawing of a plane. Figure 5.3 shows this movie clip. The sixth frame through the end contains an explosion graphic that we will use when a plane is hit.

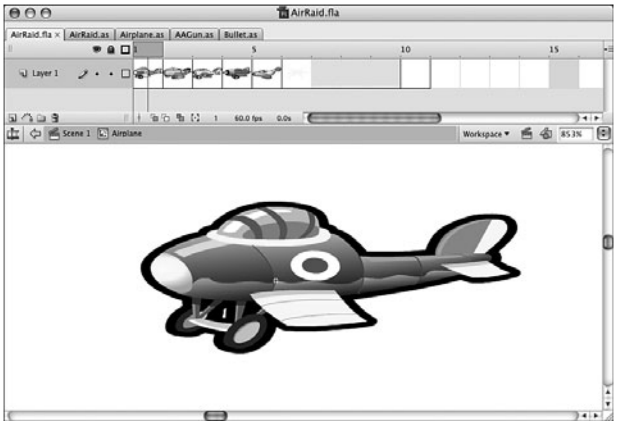


Figure 5.3
The `Airplane` *movie clip has five different airplanes, each in its own frame.*

In addition to the three class files **AAGun.as**, **Airplane.as**, and **Bullet.as**, we need a main class file for the movie, **AirRaid.as**.

Flying Airplanes

The ActionScript class for the airplanes won't be too different in structure from the `AnimatedObject` from earlier in this chapter. It will accept some parameters in the constructor function to determine the starting position and speed of the plane. It will use the time to track the difference between frames. It will use an `ENTER_FRAME` event to step forward the animation.

Class Declaration and Variables

The following code is the class definition and the variables the class will use. Because the plane will only fly horizontally, it will only need `dx`, the horizontal speed:

```
package {
    import flash.display.*;
    import flash.events.*;
    import flash.utils.getTimer;

    public class Airplane extends MovieClip {
        private var dx:Number; // speed and direction
        private var lastTime:int; // animation time
```

The Constructor Function

The constructor function will take three parameters: `side`, `speed`, and `altitude`. The `side` parameter will be either `"left"` or `"right"` depending on which side of the screen the plane will emerge from.

The `speed` parameter will be used to fill the `dx` variable. If the plane is coming from the right side of the screen, we'll automatically put a negative sign in front of the speed. So, a left-to-right plane with a `speed` of 80 will have a `dx` of 80. But, a right-to-left plane with a speed of 80 will have a `dx` of -80.

The `altitude` is just a fancy name for the vertical position of the plane. So, 0 will be the top of the screen, 50 will be 50 pixels below the top, and so on.

In addition to setting the location and `dx`, we also need to flip the plane so that it faces the right direction. We can do this by using the `scaleX` property of the movie clip. A value of –1 will flip the image:

```
public function Airplane(side:String, speed:Number, altitude:Number) {
    if (side == "left") {
        this.x = -50; // start to the left
        dx = speed; // fly left to right
        this.scaleX = -1; // reverse
    } else if (side == "right") {
        this.x = 600; // start to the right
        dx = -speed; // fly right to left
        this.scaleX = 1; // not reverse
    }
    this.y = altitude; // vertical position

    // set up animation
    addEventListener(Event.ENTER_FRAME,movePlane);
    lastTime = getTimer();
}
```

The `Airplane` function ends by setting the event timer and initializing the `lastTime` property just like we did in the `AnimatedObject` class.

Moving the Plane

The `movePlane` function first calculates the time passed, and then moves the plane according to the timer passed and the speed of the plane.

Then, it checks to see whether the plane has completed its journey across the screen. If so, the `deletePlane` function is called:

```
public function movePlane(event:Event) {
     // get time passed
     var timePassed:int = getTimer()-lastTime;
     lastTime += timePassed;

     // move plane
     this.x += dx*timePassed/1000;

     // check to see if off screen
     if ((dx < 0) && (x < -50)) {
          deletePlane();
     } else if ((dx > 0) && (x > 600)) {
          deletePlane();
     }

}
```

Removing Planes

The `deletePlane` is a somewhat self-cleaning function. You can see this in the next code block. It removes the plane from the stage with a `removeChild` command. It then removes the listener to the `movePlane` function.

NOTE

It is always a good idea to include a function with a class that deletes the object. This way, the class can handle the removal of its own listeners and any commands needed to clean up other references to itself.

For the plane to completely go away, we need to tell the main class that the plane is done. So, we start here by calling `removePlane`, a function of the main timeline's class. The main timeline is what created the plane in the first place, and in doing so will store it in an array. The `removePlane` function, which we will get to in a minute, will remove the plane from the array:

```
// delete plane from stage and plane list
public function deletePlane() {
     MovieClip(parent).removePlane(this);
     parent.removeChild(this);
     removeEventListener(Event.ENTER_FRAME,movePlane);
}
```

NOTE
After all references to an object have been reset or deleted, the Flash player will reclaim the memory used by the object.

There will be a second function for removing the airplane. This one looks similar to `deletePlane`, but it will handle the situation where the plane is hit by the player's fire. It will also kill the frame event, and tell the main class to return the plane from the array. Instead of removing the child from the stage, however, it will just tell the movie clip to go to the frame labeled "explode" and play from there.

The movie clip has an explosion graphic starting at frame 6. This goes on for a few frames, and then hits a frame with a simple `parent.removeChild(this);` and a `stop();` on it. This completes the deletion of the plane, after a brief glimpse at an explosion for the player to enjoy:

```
// plane hit, show explosion
public function planeHit() {
     removeEventListener(Event.ENTER_FRAME,movePlane);
     MovieClip(parent).removePlane(this);
     gotoAndPlay("explode");
}
```

NOTE
You can make the explosion longer by lengthening the number of frames between the "explosion" frame and the last one with the script on it. Similarly, you can place an animated explosion on those frames with no additional ActionScript needed.

Testing the Airplane Class

The main timeline is what is in charge of creating the planes, and also removing them. We'll create that class later. If we want to test the `Airplane` class, we can do it with a simple main class like this:

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

     public class AirRaid extends MovieClip {
          public function AirRaid() {
               var a:Airplane = new Airplane("right",170,30);
               addChild(a);
          }
     }
}
```

It is a good idea, if you are testing, to try different values for the parameters. For instance, try a `"left"` and a speed of 30. Try as many different values as you need to be sure that `Airplane` is working before moving on to the next class.

Moving Gun

The class that controls the anti-aircraft gun, seen in Figure 5.4, is a little different in that the movement is controlled by user actions. We could use the mouse to set the position of the gun, but that would make the game almost too easy. It only takes one flick of the wrist to move from one side of the screen to the other.

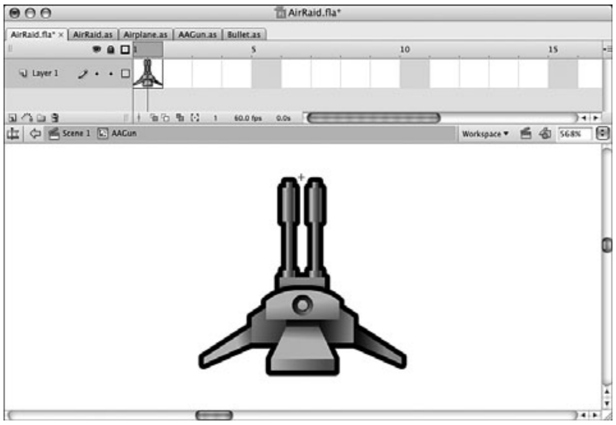


Figure 5.4
The anti-aircraft turret is positioned so its registration point is at the tips of the gun barrels.

So, instead, we'll use the left and right arrow keys to move the gun. Like the planes, it will move at a set speed to the left or right, depending on which key is pressed.

The arrow keys will actually be handled by the main movie class, not the `AAGun` class. This is because the keyboard, by default, sends events to the stage, not a particular movie clip.

The main movie class will have two variables, `leftArrow` and `rightArrow` that will be set to `true` or `false`. The `AAGun` class simply looks to those variables to see what direction, if any, to send the gun:

```
package {
    import flash.display.*;
    import flash.events.*;
    import flash.utils.getTimer;

    public class AAGun extends MovieClip {
        static const speed:Number = 150.0;
        private var lastTime:int; // animation time

        public function AAGun() {
            // initial location of gun
            this.x = 275;
            this.y = 340;

            // movement
            addEventListener(Event.ENTER_FRAME,moveGun);
        }
```

Now that the location of the gun has been set, and the listener added, the `moveGun` function runs every frame to handle the movement, if any:

```
        public function moveGun(event:Event) {
            // get time difference
            var timePassed:int = getTimer()-lastTime;
            lastTime += timePassed;

            // current position
            var newx = this.x;

            // move to the left
            if (MovieClip(parent).leftArrow) {
                newx -= speed*timePassed/1000;
            }

            // move to the right
            if (MovieClip(parent).rightArrow) {
                newx += speed*timePassed/1000;
            }

            // check boundaries
            if (newx < 10) newx = 10;
            if (newx > 540) newx = 540;

            // reposition
            this.x = newx;
        }
    }
}
```

Besides moving the gun, you can see two lines above, under the comment "check boundaries," where the new location of the gun is checked to be sure it did not go beyond the sides.

It is worth looking at how the main class handles the key presses right now. In the constructor function, two `addEventListener` calls set it up:

```
stage.addEventListener(KeyboardEvent.KEY_DOWN,keyDownFunction);
stage.addEventListener(KeyboardEvent.KEY_UP,keyUpFunction);
```

The two functions that are called set the Boolean values of `leftArrow` and `rightArrow` accordingly:

```
// key pressed
public function keyDownFunction(event:KeyboardEvent) {
    if (event.keyCode == 37) {
        leftArrow = true;
    } else if (event.keyCode == 39) {
        rightArrow = true;
    }
}
```

NOTE
The `event.keyCode` value is a number that is matched to a key on the keyboard. Key 37 is the left arrow, and key 39 is the right arrow. Key 38 and 40 are the up and down arrows, which we will use in other chapters.

```
// key lifted
public function keyUpFunction(event:KeyboardEvent) {
    if (event.keyCode == 37) {
        leftArrow = false;
    } else if (event.keyCode == 39) {
        rightArrow = false;
    }
}
```

So, the movement of the gun is really a joint effort by the main class and the `AAGun` class. The main class handles the keyboard input, and the `AAGun` class handles the movement.

There is one more part to `AAGun`, the `deleteGun` function. We'll only be using this when it is time to remove the gun from the stage to jump to the gameover frame:

```
// remove from screen and remove events
public function deleteGun() {
    parent.removeChild(this);
    removeEventListener(Event.ENTER_FRAME,moveGun);
}
```

NOTE
It is important to always remember to use `removeEventListener` to get rid of frame and timer events. Otherwise, those events will continue to happen even after you think you have deleted the parent object.

Skyward Bullets

Bullets are probably the simplest of all the moving objects. In this game, the graphic is actually a grouping of bullets as seen in Figure 5.5.

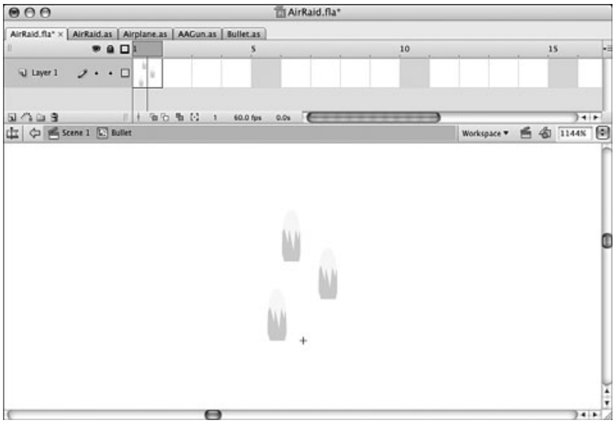


Figure 5.5
The bullet grouping has the registration point at the bottom, so when they start at the gun, they are just above the tops of the barrels.

They will originate at the location of the gun, and move directly upward until they reach the top of the screen. All the code in the `Bullet` class we have seen before in the `Airplane` and `AAGun` classes.

The constructor function will accept a starting `x` and `y` value, and a speed. The speed will be applied vertically, rather than horizontally like the airplanes:

```
package {
    import flash.display.*;
    import flash.events.*;
    import flash.utils.getTimer;

    public class Bullet extends MovieClip {
        private var dy:Number; // vertical speed
        private var lastTime:int;

        public function Bullet(x,y:Number, speed: Number) {
            // set start position
            this.x = x;
            this.y = y;
            // get speed
            dy = speed;
            // set up animation
            lastTime = getTimer();
            addEventListener(Event.ENTER_FRAME,moveBullet);
        }

        public function moveBullet(event:Event) {
            // get time passed
            var timePassed:int = getTimer()-lastTime;
            lastTime += timePassed;

            // move bullet
            this.y += dy*timePassed/1000;

            // bullet past top of screen
            if (this.y < 0) {
                deleteBullet();
            }
        }
```

The `removeBullet` function, like the `removePlane` function, will exist in the main class. It will be in charge of removing bullets when they reach the top of the screen:

```
        // delete bullet from stage and plane list
        public function deleteBullet() {
            MovieClip(parent).removeBullet(this);
            parent.removeChild(this);
            removeEventListener(Event.ENTER_FRAME,moveBullet);
        }
    }
}
```

To start a bullet, the player presses the spacebar. We need to modify the `keyDownFunction` in the `AirRaid` class to accept spaces and pass them along to a function that handles the creation of a new `Bullet`:

```
// key pressed
public function keyDownFunction(event:KeyboardEvent) {
    if (event.keyCode == 37) {
        leftArrow = true;
    } else if (event.keyCode == 39) {
        rightArrow = true;
    } else if (event.keyCode == 32) {
        fireBullet();
    }
}
```

NOTE

The key code 32 is for the spacebar. To find out what other keys correspond to which codes, look in the Flash help. Search for "Keyboard Keys and Key Code Values."

The `fireBullet` function passes the location of the gun and a speed to the new `Bullet`. It will also add the new `Bullet` to the array `bullets` so that it can keep track of it later for collision detection:

```
public function fireBullet() {
     var b:Bullet = new Bullet(aagun.x,aagun.y,-300);
     addChild(b);
     bullets.push(b);
}
```

Now that we have airplanes, an anti-aircraft gun, and `Bullet` objects, it is time to tie all of these together with the main `AirRaid` class.

The Game Class

The `AirRaid` class contains all the game logic. It is here that we will create the initial game objects, check for collisions, and handle scoring. After the game gets going, it will look something like Figure 5.6.

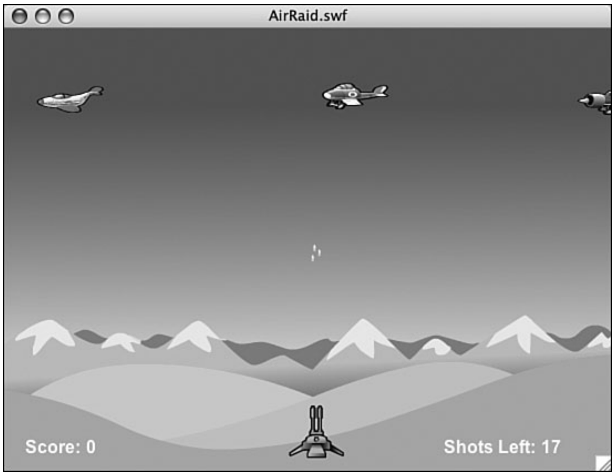


Figure 5.6
The Air Raid game with anti-aircraft gun, bullet in mid-movement, and two airplanes flying overhead.

Class Declaration

The class needs the standard classes we have been using so far, including access to `getTimer` and text fields:

```
package {
     import flash.display.*;
     import flash.events.*;
     import flash.utils.Timer;
     import flash.text.TextField;
```

The variables we need for the class include references to the gun, and arrays that reference the airplanes and bullets that we'll create:

```
     public class AirRaid extends MovieClip {
          private var aagun:AAGun;
          private var airplanes:Array;
          private var bullets:Array;
```

The next two variables are `true` or `false` values that track the player's use of the left- and right-arrow keys. These need to be `public` variables because the `aagun` object will be accessing them to determine whether to move:

```
public var leftArrow, rightArrow:Boolean;
```

NOTE

You can include more than one variable on a variable definition line. This works great when you've got small groups of variables that are related and of the same type. The `leftArrow` and `rightArrow` variables are a good example.

The next variable, `nextPlane`, is going to be a `Timer`. We'll use this to determine when the next plane will appear.

```
private var nextPlane:Timer;
```

Finally, we've got two score-keeping variables. The first keeps track of how many shots are left, and the second how many hits the player scored:

```
private var shotsLeft:int;
private var shotsHit:int;
```

There is no `AirRaid` constructor function for this game because it is not starting on the first frame. Instead, we'll have one called `startAirRaid` that will be called from the main timeline on the "play" frame.

The function starts off by setting the number of shots left to 20 and the score to 0:

```
public function startAirRaid() {
    // init score
    shotsLeft = 20;
    shotsHit = 0;
    showGameScore();
```

Next, the anti-aircraft gun is created and added to the stage, as `aagun`:

```
    // create gun
    aagun = new AAGun();
    addChild(aagun);
```

We also need to start off the arrays to hold the bullets and airplanes:

```
    // create object arrays
    airplanes = new Array();
    bullets = new Array();
```

To know which arrow keys have been pressed, we need two listeners, one for the key down events, one for the key up events:

```
    // listen for keyboard
    stage.addEventListener(KeyboardEvent.KEY_DOWN,keyDownFunction);
    stage.addEventListener(KeyboardEvent.KEY_UP,keyUpFunction);
```

We need an `ENTER_FRAME` event listener to trigger the `checkForHits` function. This will be the all-important collision detection between the bullets and the planes:

```
    // look for collisions
    addEventListener(Event.ENTER_FRAME,checkForHits);
```

Now, we need to kick off the game by getting some planes in the air. The `setNextPlane` function will do this, and we look at it next:

```
    // start planes flying
    setNextPlane();
}
```

Creating a New Plane

New planes will need to be created periodically, at somewhat random times. To do this, we'll use a `Timer` and trigger a `newPlane` function call in the near future. The `setNextPlane` function will create the `Timer` with only one event, and set it for one to two seconds in the future:

```
public function setNextPlane() {
    nextPlane = new Timer(1000+Math.random()*1000,1);
    nextPlane.addEventListener(TimerEvent.TIMER_COMPLETE,newPlane);
    nextPlane.start();
}
```

When the `Timer` is done, it will call `newPlane` to create a new airplane and send it on its way. The three parameters of the `Airplane` object are randomly decided using several `Math.random()` function results. Then, the plane is created and added to the stage. It is also added to the `airplanes` array.

Finally, the `setNextPlane` function is called again to schedule the next plane:

```
public function newPlane(event:TimerEvent) {
    // random side, speed and altitude
    if (Math.random() > .5) {
        var side:String = "left";
    } else {
        side = "right";
    }
    var altitude:Number = Math.random()*50+20;
    var speed:Number = Math.random()*150+150;

    // create plane
    var p:Airplane = new Airplane(side,speed,altitude);
    addChild(p);
    airplanes.push(p);

    // set time for next plane
    setNextPlane();
}
```

Collision Detection

The most interesting function in this entire game is the `checkForHits` function. It looks at all the bullets and all the airplanes and determine whether any of them are intersecting at the moment.

NOTE

Notice that we are looping backward through the arrays. This is so that when we delete an item from an array we don't trip over ourselves. If we were moving forward through the array, we could delete item 3 in the array, which would make the old item 4, the new item 3. Then, moving forward to look at item 4, we would be skipping an item.

We'll use the `hitTestObject` to see whether the bounding boxes of the two movie clips overlap. If they do, we'll do several things. First, we'll call `planeHit`, which will start the death of the airplane. Then, we'll delete the bullet. We'll up the number of hits and redisplay the game score. Then, we'll stop looking at collisions for this airplane and move on to the next bullet in the list:

```
// check for collisions
public function checkForHits(event:Event) {
     for(var bulletNum:int=bullets.length-1;bulletNum>=0;bulletNum--){
          for (var airplaneNum:int=airplanes.length-1;airplaneNum>=0;airplaneNum--) {
               if (bullets[bulletNum].hitTestObject(airplanes[airplaneNum])) {
                    airplanes[airplaneNum].planeHit();
                    bullets[bulletNum].deleteBullet();
                    shotsHit++;
                    showGameScore();
                              break;
               }
          }
     }

     if ((shotsLeft == 0) && (bullets.length == 0)) {
          endGame();
     }
}
```

At the end of the function, we'll check to see whether the game is over. This happens when there are no shots left, and the last bullet has gone off the top of the screen or it has hit a plane.

Handling Keyboard Input

The next two functions handle the key presses. We've seen both these functions before:

```
// key pressed
public function keyDownFunction(event:KeyboardEvent) {
    if (event.keyCode == 37) {
        leftArrow = true;
    } else if (event.keyCode == 39) {
        rightArrow = true;
    } else if (event.keyCode == 32) {
        fireBullet();
    }
}

// key lifted
public function keyUpFunction(event:KeyboardEvent) {
    if (event.keyCode == 37) {
        leftArrow = false;
    } else if (event.keyCode == 39) {
        rightArrow = false;
    }
}
```

To create a new bullet when the player presses the spacebar, just simply create the object and feed it the location of the gun and the speed of the bullet (this case, 300 pixels per second).

We add the bullet to the `bullets` array and subtract one from `shotsLeft`. We'll also update the game score.

Notice that before any of this happens, we'll check `shotsLeft` to make sure the player can even fire another shot. This prevents the player from getting a few extra bullets in at the end of the game:

```
// new bullet created
public function fireBullet() {
    if (shotsLeft <= 0) return;
    var b:Bullet = new Bullet(aagun.x,aagun.y,-300);
    addChild(b);
    bullets.push(b);
    shotsLeft--;
    showGameScore();
}
```

Other Functions

We have now called `showGameScore` a few times. This little function just places the `shotsHit` and `shotsLeft` into text fields on the stage. These aren't text fields that we created in the code, but rather ones that I put on the stage manually in the sample movie. I didn't want to clutter this example up with `TextField` and `TextFormat` code:

```
public function showGameScore() {
     showScore.text = String("Score: "+shotsHit);
     showShots.text = String("Shots Left: "+shotsLeft);
}
```

NOTE

Even though I didn't create the text fields in the code, I still needed to put the `import flash.text.TextField;` statement at the start of the class. You need this to access text fields and to create them.

The next two functions simply remove a single item from an array. The `for…in` loop is used to go through the array, and then the `splice` command is used to remove it when it is found. The `break` command is used to quit looping after the match has been found.

We need a function to remove a plane from the `airplanes` array, and another to remove a bullet from the `bullets` array:

```
// take a plane from the array
public function removePlane(plane:Airplane) {
     for(var i in airplanes) {
          if (airplanes[i] == plane) {
               airplanes.splice(i,1);
               break;
          }
     }
}

// take a bullet from the array
public function removeBullet(bullet:Bullet) {
     for(var i in bullets) {
          if (bullets[i] == bullet) {
               bullets.splice(i,1);
               break;
          }
     }
}
```

We could use a single function to take the place of both the `removePlane` and `removeBullet` function. This single function would get passed into it both the array and the item to find. By using separate functions, however, we set up future development of the game where removing airplanes and bullets might have other effects. For instance, removing an airplane could be the signal to call `setNewPlane` instead of that happening right after a plane is created.

Cleaning Up after the Game

When a game ends, there are still game elements on the screen. We know all the bullets are gone because that is a condition that has to be met for the game to be over. However, planes and the gun are still there.

We didn't store all the display objects in a single array, as we did for the deduction game in the preceding chapter. Instead, we've got them in the `airplanes` array, the `aagun` variable, and the `bullets` array, which we know is already empty.

After removing the airplanes and the gun, we also need to remove the keyboard listeners and the `checkForHits` event listener. The `nextPlane Timer` needs to be cleaned up, too. Then, we can go to the gameover frame without any of the game elements hanging around:

```
// game is over, clear movie clips
public function endGame() {
    // remove planes
    for(var i:int=airplanes.length-1;i>=0;i--) {
        airplanes[i].deletePlane();
    }
    airplanes = null;

    aagun.deleteGun();
    aagun = null;

    stage.removeEventListener(KeyboardEvent.KEY_DOWN,keyDownFunction);
    stage.removeEventListener(KeyboardEvent.KEY_UP,keyUpFunction);
    removeEventListener(Event.ENTER_FRAME,checkForHits);

    nextPlane.stop();
    nextPlane = null;

    gotoAndStop("gameover");
}
```

After this function, you'll need two more brackets to close out the class and the package.

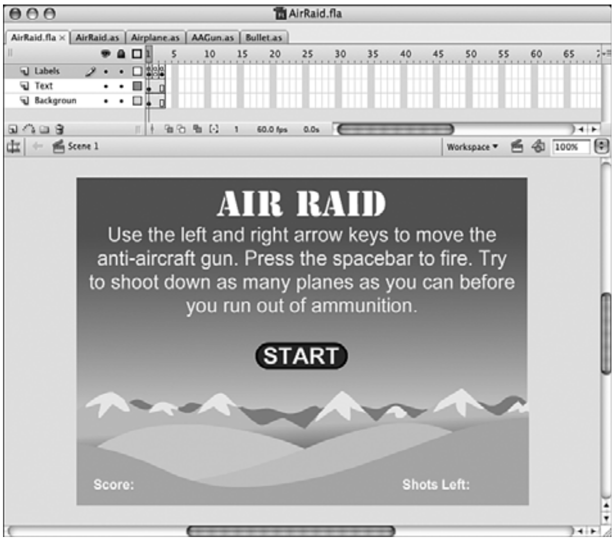
One good thing about using manually created text fields in the main timeline rather than ActionScript-created fields is that they will stick around for the gameover frame. This means that the player can see their score on that last frame.

Modifying the Game

The **AirRaid.fla** movie contains the same frame scripts and buttons that **Deduction.fla** from the preceding chapter had. The intro frame has a Start button on it, and the gameover frame has a Play Again button. The middle frame is labeled `"play"`.

In this game, I've also added instructions to the intro frame. Figure 5.7 shows the first frame complete with the instructions, title, Start button, and the text fields at the bottom that will be used in the game.

Figure 5.7
The intro frame has instructions and a Start button.



Use the left and right arrow keys to move the anti-aircraft gun. Press the spacebar to fire. Try to shoot down as many planes as you can before you run out of ammunition.

Improving this game can be as simple as adding more planes, or planes that follow a more serious style. The background and turret can also be changed.

In the code, you can vary the speed at which the planes appear, and how fast they move. Perhaps you might even want to have these speeds increase with time.

You can also, of course, change the number of bullets that the player starts with.

More drastic modifications can include changing the theme of the game entirely. You can revert back to the old submarine games by making the planes into ships and the gun into a periscope viewer. In that case, I would slow down the speed of the bullets dramatically and use the background art to create some depth to the scene.

Paddle Ball

SOURCE FILES
http://flashgameu.com
A3GPU05_PaddleBall.zip

Air Raid involved simple one-dimensional movement for a variety of objects and collisions resulting in objects being eliminated. This next game, Paddle Ball, includes diagonal movement, and collisions that result in bounces.

Paddle Ball is a version of Breakout, an early and popular video arcade game. Versions of this game appear frequently online, and even come standard on many cell phones and the iPod.

NOTE
The original version of Breakout for Atari in 1976 was developed by Steve Jobs and Steve Wozniak, before they founded Apple Computer. Wozniak's chip compact design for the game didn't work with Atari's manufacturing process, and so it was scrapped.

Versions of Breakout appeared as hidden Easter eggs in many versions of the Mac OS. Even today, it is included on most iPods.

In this version of Paddle Ball, the player controls a paddle at the bottom of the screen, which he or she can move left and right with the mouse. The main active element in the game is a ball, which can bounce off the side, walls, and top, but passes through the bottom of the screen if the paddle isn't there.

At the top of the screen are many rectangular bricks that the player must eliminate by directing the ball to them with the paddle.

Setting Up the Movie

The movie is arranged just like Air Raid and Deduction. The first frame will be the intro, the third frame gameover. They will both have buttons on them to start a new game and instructions on the first frame.

The middle frame will be play. This is where the game play will take place. On it, we've got a drawn border, a text field in the middle for messages such as "Click to Start," and a text field to the bottom right to tell players how many balls they have left. Figure 5.8 shows you these three elements against a dark background.

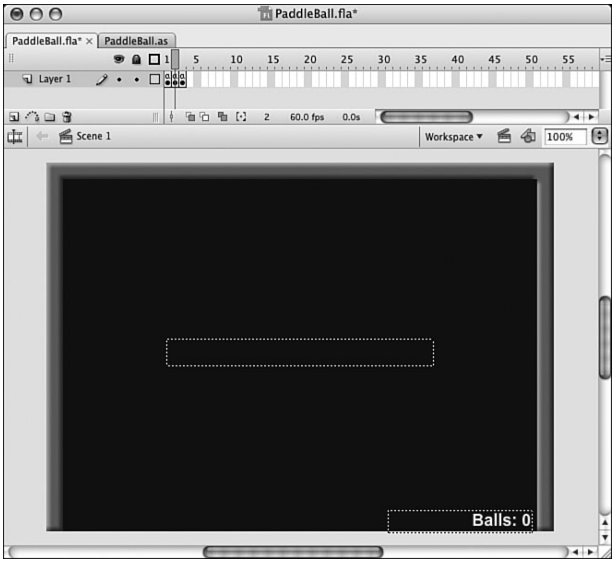
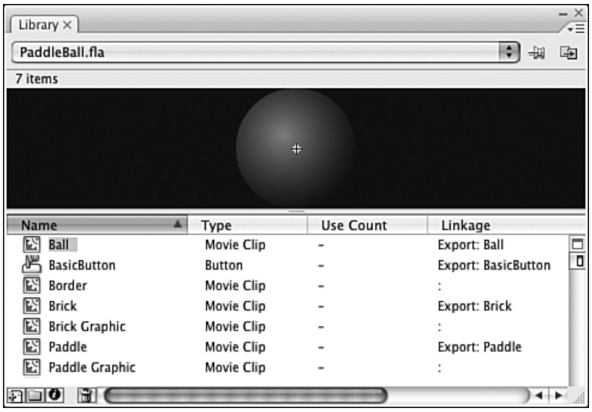


Figure 5.8
The middle text field is "gameMessage," and the one in the lower right is "ballsLeft."

The library in this movie is also a little more crowded than the games we have attempted so far. Figure 5.9 shows the library, complete with the class name of the movie clips that our scripts will be accessing.



Name	Type	Use Count	Linkage
Ball	Movie Clip	-	Export: Ball
BasicButton	Button	-	Export: BasicButton
Border	Movie Clip	-	:
Brick	Movie Clip	-	Export: Brick
Brick Graphic	Movie Clip	-	:
Paddle	Movie Clip	-	Export: Paddle
Paddle Graphic	Movie Clip	-	:

Figure 5.9
There are seven items in the library, including the ball, brick, and paddle.

You'll notice that there is a `"Brick"` and a `"Brick Graphic"` as well as a `"Paddle"` and `"Paddle Graphic"`. The second of each pair is contained in the first. So, the `"Brick Graphic"` is the one and only element in `"Brick"`.

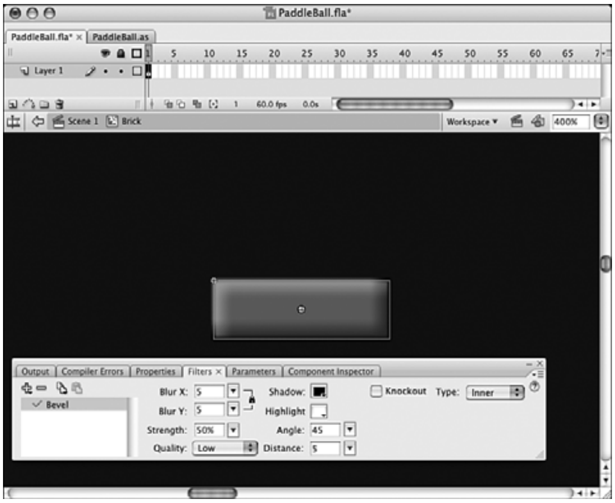
The reason for this is that we can easily apply a filter to these elements. A filter, like the Bevel filter we will be using here, can be applied in the Properties panel of a movie clip. However, because both `"Brick"` and `"Paddle"` will be created by ActionScript, we can't apply filters to them this way. So, we have a "`Brick Graphic"` that is inside `"Brick"`.

The `"Brick Graphic"` will have the filter applied to it. Then, we can create a copy of `"Brick"` in ActionScript without worrying about this.

NOTE

We could apply the filter in ActionScript, too. However, this would take extra code that doesn't really have anything to do with the game. Another good reason to leave the filter settings outside of ActionScript is that these things can be left to an artist who is working along with the programmer. It is easier for artists to create the graphics and apply the filters than to rely on the programmer to do it for them.

Figure 5.10 shows the `Brick` movie clip, with the `Brick Graphic` movie clip inside it. You can also see the Properties panel with the filter settings.



Blur X: 5
Blur Y: 5
Strength: 50%
Quality: Low
Shadow:
Highlight
Angle: 45
Distance: 5
Knockout
Type: Inner

Figure 5.10
A Bevel filter is used to make simple rectangles into more interesting graphics.

Class Definition

Unlike the Air Raid game, this game uses one single ActionScript class to control everything. This class will need a lot of support, including `getTimer`, the `Rectangle` class, and text fields:

```
package {
    import flash.display.*;
    import flash.events.*;
    import flash.utils.getTimer;
    import flash.geom.Rectangle;
    import flash.text.TextField;
```

There are a lot of constants to consider in this game. The following list is pretty self-explanatory. It includes the positions and dimensions of many of the items in the game, such as the ball, walls, and paddle:

```
public class PaddleBall extends MovieClip {
    // environment constants
    private const ballRadius:Number = 9;
    private const wallTop:Number = 18;
    private const wallLeft:Number = 18;
    private const wallRight:Number = 532;
    private const paddleY:Number = 380;
    private const paddleWidth:Number = 90;
    private const ballSpeed:Number = .2;
    private const paddleCurve:Number = .005;
    private const paddleHeight:Number = 18;
```

The only two objects that move in this game are the ball and paddle. In addition, we need an array to store the bricks:

```
        // key objects
        private var paddle:Paddle;
        private var ball:Ball;

        // bricks
        private var bricks:Array;
```

To keep track of the ball's velocity, we'll use two variables: `ballDX` and `ballDY`. We also need a `lastTime` variable as in Air Raid:

```
        // ball velocity
        private var ballDX:Number;
        private var ballDY:Number;

        // animation timer
        private var lastTime:uint;
```

NOTE

Velocity is a combined measurement of speed and direction. A variable such as `dx` measures the horizontal speed of an object. But a combination of variables, such as `dx` and `dy`, is said to measure both speed and direction: velocity. Alternatively, a game may track speed as one variable (pixels per second) and direction as another (a vector or angle). Combined, these also represent velocity.

One last variable is the number of balls remaining. This is the first game we have built that gives the player a number of chances, or lives, before the game is over. The player has three balls to use when playing. When the player misses the third ball and lest it pass the paddle, the game ends:

```
        // number of balls left
        private var balls:Number;
```

There will be no constructor function for this game because we'll wait for the second frame to start. So, we leave out a `PaddleBall` function.

Starting the Game

When the game is started, the paddle, bricks, and ball are created. The creation of the pattern of bricks is delegated to another function. We look at that next.

The number of balls is set at three, and the initial game message is placed in the text field. Also, the `lastTime` is set to zero. This is different from how we have worked before, setting it to `getTimer`. I explain this when we get to the animation functions that use `lastTime`.

Two listeners are set. The first one just calls the `moveObjects` function every frame. The second is an event listener placed on the stage to capture mouse clicks. We'll be asking the player to "Click to Start," so we need to get this click and use it by running `newBall`:

```
public function startPaddleBall() {

     // create paddle
     paddle = new Paddle();
     paddle.y = paddleY;
     addChild(paddle);

     // create bricks
     makeBricks();

     balls = 3;
     gameMessage.text = "Click To Start";

     // set up animation
     lastTime = 0;
     addEventListener(Event.ENTER_FRAME,moveObjects);
     stage.addEventListener(MouseEvent.CLICK,newBall);
}
```

The `makeBricks` function creates a grid of bricks. There will be eight columns across and five rows down. We will use a nested loop with `x` and `y` variables to create all 40 bricks. The positions of each brick are spaces 60 pixels vertically and 20 horizontally, with a 65- and 50-pixel offset.

```
// make collection of bricks
public function makeBricks() {
     bricks = new Array();

     // create a grid of bricks, 5 vertical by 8 horizontal
     for(var y:uint=0;y<5;y++) {
            for(var x:uint=0;x<8;x++) {
                var newBrick:Brick = new Brick();
                // space them nicely
                newBrick.x = 60*x+65;
                newBrick.y = 20*y+50;
                addChild(newBrick);
                bricks.push(newBrick);
            }
        }
    }
```

NOTE

When creating arrangement functions like this, don't be afraid to experiment with the numbers to get the desired result. For instance, I plugged numbers into the `makeBricks` function until I got an arrangement of bricks that looked good. Sure, I could have calculated the spacing and offsets in my head or on paper beforehand, but it was easier and quicker to just make a few educated guesses.

ActionScript is a great environment for experimentation and discovery. You don't need to plan every little thing out before typing the first line of code.

One of the advantages of farming out the brick creation to its own function is that you can later replace it with a function that produces different patterns of bricks. It can even read from a database of brick layouts if you want. Any changes you want to make to the pattern are isolated in a single call to `makeBricks`, so it is even easy to have a second programmer work on brick arrangement, while you work on game play.

Figure 5.11 shows the game at the start, with the ball, paddle, and bricks. You can also see the walls, which are purely cosmetic. The ball will be bouncing off the invisible walls we created by setting `wallLeft`, `wallRight`, and `wallTop`.

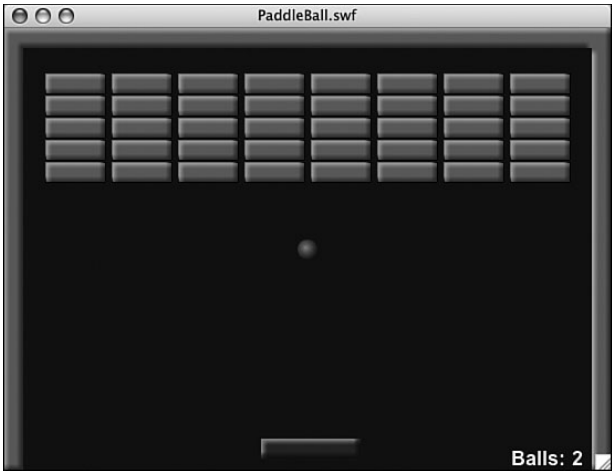


Figure 5.11
All the game elements are shown at the very start of the game.

Starting a New Ball

When the game begins, there is no ball. Instead, the message "Click to Start" appears, and the player must click the stage. This will call `newBall`, which creates a new `Ball` object and sets its position and related properties.

First, `newBall` checks to make sure that `ball` is `null`. This prevents the user from clicking on the screen while a ball is already in play.

Next, the `gameMessage` text field is cleared:

```
public function newBall(event:Event) {
     // don't go here if there is already a ball
     if (ball != null) return;

     gameMessage.text = "";
```

A new ball is created at the exact center of the screen, using the halfway point between `wallLeft` and `wallRight`, and `wallTop` and the vertical position of the paddle:

```
     // create ball, center it
     ball = new Ball();
     ball.x = (wallRight-wallLeft)/2+wallLeft;
     ball.y = 200;//(paddleY-wallTop)/2+wallTop;
     addChild(ball);
```

The ball velocity is set to be straight down, at the `ballSpeed` constant:

```
     // ball velocity
     ballDX = 0;
     ballDY = ballSpeed;
```

The `balls` variable is reduced by one, and the text field at the bottom right is changed to show the number remaining. Also, `lastTime` is reset to zero so that the animation timekeeper starts off fresh:

```
     // use up a ball
     balls--;
     ballsLeft.text = "Balls: "+balls;

     // reset animation
     lastTime = 0;
}
```

The `newBall` function will be used at the start of the game, and also to begin a new ball in the middle of the game.

Game Animation and Collision Detection

So far, the game code has been pretty simple and straightforward. When we start looking at the moving objects, however, things get complicated. The ball must detect collisions with both the paddle and the bricks. It then needs to react to the collisions appropriately.

The event listener for `ENTER_FRAME` calls `moveObjects` every frame. This function then delegates to two other functions, `movePaddle` and `moveBall`:

```
public function moveObjects(event:Event) {
     movePaddle();
     moveBall();
}
```

Paddle Movement

The `movePaddle` function is simple. It just sets the `x` property of the paddle to the `mouseX` location. However, it also uses `Math.min` and `Math.max` to restrict the location to the left and right sides of the stage.

NOTE

The `mouseX` and `mouseY` properties return the cursor location relative to the current display object. In this case, that would be the main class, which is equivalent to the stage. If we were looking at `mouseX` and `mouseY` from inside a movie clip class, we'd need to adjust the results, or look for `stage.mouseX` and `stage.mouseY`.

```
public function movePaddle() {
     // match horizontal value with the mouse
     var newX:Number = Math.min(wallRight-paddleWidth/2,
          Math.max(wallLeft+paddleWidth/2,
               mouseX));
     paddle.x = newX;
}
```

Ball Movement

The function that moves the ball, `moveBall`, gets the lion's share of code. The basic movement is similar to the moving objects in Air Raid. However, the collisions are far more complex.

The function starts off with a check of `ball` to make sure it is not `null`. If it is, we are between balls, and the rest can be skipped:

```
public function moveBall() {
     // don't go here if in between balls
     if (ball == null) return;
```

Remember that we initialized the `lastTime` variable to zero rather than `getTimer`. This is so that the time it takes to create the game objects and draw the screen for the first time is not used in determining the amount of the first animation step. For instance, if it takes 500 milliseconds for the game to start, `getTimer()` minus `lastTime` will be 500 milliseconds or greater. So, the ball will jump quite a bit before the player has a chance to even react.

NOTE

One reason this game takes a half a second or so to begin is the use of Bevel filters. This slows down the start of the game as the modifications to the visual representation of the bricks and paddle are generated. However, it will not slow down the game play after that initial hit.

By starting the `lastTime` at zero, we can recognize it here in the animation function and get a fresh new value for `lastTime`. This means that the first time through the function the `timePassed` will most likely be zero. But that won't affect anything. What it does do, however, is make sure the animation timer isn't ticking until we are at the point of calling `moveBall` for the first time:

```
// get new location for ball
if (lastTime == 0) lastTime = getTimer();
var timePassed:int = getTimer()-lastTime;
lastTime += timePassed;
var newBallX = ball.x + ballDX*timePassed;
var newBallY = ball.y + ballDY*timePassed;
```

Collision Detection

To start our collision detection, we'll get the rectangle of the ball. In fact, we'll get two different versions of the rectangle: the current ball rectangle, called `oldBallRect`; and the rectangle of the ball if it were to complete its movement unimpeded as `newBallRect`.

NOTE

The `Rectangle` object is a good example of an object that works to bring out more information from the data you feed it. You give it an `x` and `y` position and a width and a height. But, you can ask it for interpreted information like the top, bottom, left, and right sides of the rectangle. You can even get `Point` objects for the corners (for example, `bottomRight`). We'll be using the top, bottom, left, and right sides of the rectangle in our calculations.

The way we calculate `oldBallRect` and `newBallRect` is to use the `x` and `y` positions, plus or minus the `ballRadius`. For instance, `ball.x-ballRadius` gives us the `x` position, and `ballRadius*2` gives us the width. We calculate the `paddleRect` the same way:

```
var oldBallRect = new Rectangle(ball.x-ballRadius,
    ball.y-ballRadius, ballRadius*2, ballRadius*2);
var newBallRect = new Rectangle(newBallX-ballRadius,
    newBallY-ballRadius, ballRadius*2, ballRadius*2);
var paddleRect = new Rectangle(paddle.x-paddleWidth/2,
    paddle.y-paddleHeight/2, paddleWidth, paddleHeight);
```

Now that we have these three rectangles at our fingertips, we can use them to see whether the ball has hit paddle. This will happen when the bottom of the ball passes the top of the paddle. But, determining this moment is harder than it seems. We don't want to simply know whether the bottom of the ball is greater than the bottom of the paddle. We want to know that this has just happened, right now in this step of the animation. So, the correct question to ask is this: Is the bottom of the ball greater than the top of the paddle, *and* was the bottom of the ball just previously above the top of the paddle? If both of these conditions are met, the ball has just now passed the paddle. See Figure 5.12 for more clarification.

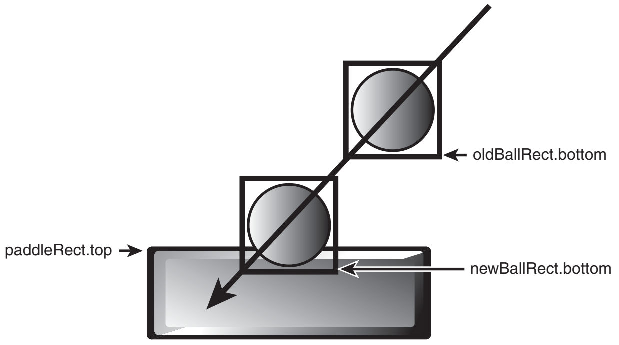


Figure 5.12
The diagram shows the ball in the previous frame, and the ball as it would be if the motion were unimpeded. The ball has crossed into the paddle and must be deflected.

Another test that needs to be performed is whether the horizontal location of the ball matches that of the paddle. So, if the right side of the ball is greater than the left side of the paddle, *and* the left side of the ball is less than the right side of the paddle, we have an actual collision.

In case of a collision, the ball must be deflected upward. This is as easy as reversing the direction of `ballDY`. In addition, a new location for the ball must be defined. After all, the ball cannot remain inside the paddle as it is now, as shown in Figure 5.12.

So, the distance past the top of the paddle is calculated, and the ball is reflected up by twice that distance (see Figure 5.13).

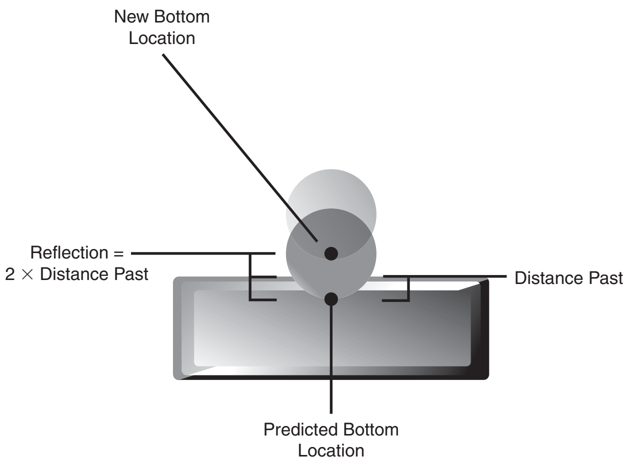


Figure 5.13
The ball moves slightly into the paddle, so it is placed the same distance back away from the paddle.

```
// collision with paddle
if (newBallRect.bottom >= paddleRect.top) {
    if (oldBallRect.bottom < paddleRect.top) {
        if (newBallRect.right > paddleRect.left) {
            if (newBallRect.left < paddleRect.right) {
                // bounce back
                newBallY -= 2*(newBallRect.bottom - paddleRect.top);
                ballDY *= -1;
                // decide new angle
                ballDX = (newBallX-paddle.x)*paddleCurve;
            }
        }
```

While the vertical speed of the ball is simply reflected, the horizontal speed, `ballDX`, is set to a new value altogether. This value is determined by the distance from the center of the paddle. It is multiplied by a constant `paddleCurve`.

The idea here is that the player needs to direct the ball. If the player is just hitting it with an actual flat surface, the ball will never shoot off at any angle except the one it started with. The game would most likely be impossible to win.

The effect of this system is that the ball will bounce straight up and down at the center of the paddle and shoot off at steeper and steeper angles toward the ends.

NOTE

One common way to represent this visually is to have a slight curve at the top of the paddle. This gives the player the general idea of what would happen when the ball hits the paddle. However, because this behavior has been in every Breakout-style game since the first, most people take it for granted.

If the ball has passed the paddle, there is no turning back. But, we don't remove the ball from play right away. Instead, it is allowed to continue until it reaches the bottom of the screen. Then, it is removed.

If this is the last ball, the game is over, and `endGame` is called. Otherwise, the `gameMessage` field is populated with "Click For Next Ball." Because the `ball` variable is set to `null`, the `moveBall` function no longer acts on anything, and the `newBall` function accepts the next click as a trigger to create a new ball. We can, and should, quit out of this function now with a `return` command. No need to check for wall collisions or brick collisions if the ball is gone:

```
        } else if (newBallRect.top > 400) {
            removeChild(ball);
            ball = null;
            if (balls > 0) {
                gameMessage.text = "Click For Next Ball";
            } else {
                endGame();
            }
            return;
        }
    }
```

Next, we check for collisions with the three walls. These are simpler checks because the ball should never go past one of these. In each case, the vertical or horizontal speed is reversed, and the location of the ball is altered in the same way we did with the paddle collisions, so the ball is never "in" the walls:

```
    // collision with top wall
    if (newBallRect.top < wallTop) {
        newBallY += 2*(wallTop - newBallRect.top);
        ballDY *= -1;
    }

    // collision with left wall
    if (newBallRect.left < wallLeft) {
        newBallX += 2*(wallLeft - newBallRect.left);
        ballDX *= -1;
    }

    // collision with right wall
    if (newBallRect.right > wallRight) {
        newBallX += 2*(wallRight - newBallRect.right);
        ballDX *= -1;
    }
```

To calculate the collision with bricks, we need to loop through all the bricks and check each one. For each, we create a `brickRect` so that we can access the top, bottom, left, and right of the brick just as easily as we can with the ball.

NOTE

Normally when you want to loop through an array of objects looking for a collision, you do it in reverse. This way, you can remove objects in the list without skipping any. This time, however, we can move forward because after the ball collides with a brick, we stop looking for other collisions (because only one collision should be happening at a time).

Detecting a collision with a brick is easy, but reacting to it is trickier. Because we have a `Rectangle` for the ball, and one for the brick, we can use the `intersects` function to see whether the new location of the ball is inside a brick.

If so, it is a matter of determining what side of the brick was hit. A series of tests compares the sides of the ball with the opposite sides of the bricks. When a crossover side has been found, the ball is reflected in the correct direction, and the location is adjusted:

```
// collision with bricks
for(var i:int=bricks.length-1;i>=0;i--) {

    // get brick rectangle
    var brickRect:Rectangle = bricks[i].getRect(this);

    // is there a brick collision
    if (brickRect.intersects(newBallRect)) {

        // ball hitting left or right side
        if (oldBallRect.right < brickRect.left) {
            newBallX += 2*(brickRect.left - oldBallRect.right);
            ballDX *= -1;
        } else if (oldBallRect.left > brickRect.right) {
            newBallX += 2*(brickRect.right - oldBallRect.left);
            ballDX *= -1;
        }

        // ball hitting top or bottom
        if (oldBallRect.top > brickRect.bottom) {
            ballDY *= -1;
            newBallY += 2*(brickRect.bottom-newBallRect.top);
        } else if (oldBallRect.bottom < brickRect.top) {
            ballDY *= -1;
            newBallY += 2*(brickRect.top - newBallRect.bottom);
        }
```

If the ball has collided with a brick, the brick should be removed. In addition, if the bricks array is empty, the game is over. We also want to use return here because if the game is over, there is no need to set the ball location at the end of this function or check for other collisions:

```
                // remove the brick
                removeChild(bricks[i]);
                bricks.splice(i,1);
                if (bricks.length < 1) {
                    endGame();
                    return;
                }
            }
        }

        // set new position
        ball.x = newBallX;
        ball.y = newBallY;
    }
```

One important aspect of this game is that there are really two game modes: The first is with the ball in play, and the second is waiting for the player to click the screen to create a new ball. The code tells the difference between these by looking at the value of ball. If it is null, the game is in the second state.

Game Over

The game ends when one of two things happens: The player loses the last ball, or the last brick is hit.

Just like in Air Raid, we use the endGame function to clean up all the leftover movie clips. We also set references to these movie clips to null so that the Flash player can remove them from memory.

It is important to check to make sure the ball isn't already gone because it will be if the endGame function is called when the last ball is lost.

We also need to remove the listeners, both the one that calls moveObjects every frame and the one that listens for mouse clicks:

```
    function endGame() {
        // remove paddle and bricks
        removeChild(paddle);
        for(var i:int=bricks.length-1;i>=0;i--) {
            removeChild(bricks[i]);
        }
        paddle = null;
        bricks = null;
        // remove ball
        if (ball != null) {
            removeChild(ball);
            ball = null;
        }


        // remove listeners
        removeEventListener(Event.ENTER_FRAME,moveObjects);
        stage.removeEventListener(MouseEvent.CLICK,newBall);

        gotoAndStop("gameover");
    }
```

At the end of the code, don't forget the closing brackets to close off the class and the package.

Modifying the Game

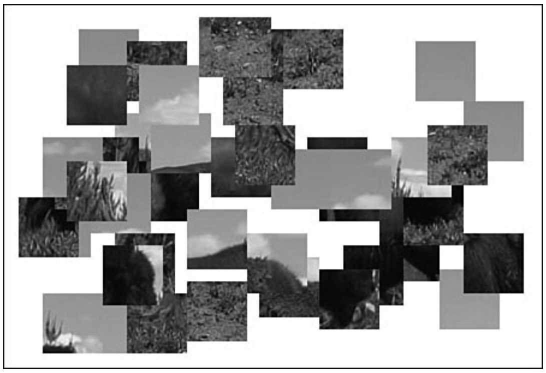
This game cries out for sound. You can add sound to this game very easily using the examples from the matching game in Chapter 3, "Basic Game Framework: A Matching Game." One for hitting the paddle and another for hitting a brick is a good start. Also, one for hitting the walls and another for missing the ball would be nice.

Another great modification is to have different colored bricks. You could do this by using multiple frames in the `Brick Graphic` movie clip, and then going to the appropriate frame. You could make each row a different color.

Scoring is a good idea, although progress in the game is obvious without it. Scoring works better if you create multiple levels. More levels could be done by simply increasing the speed of the ball with each level, or presenting a new arrangement of bricks.

When the player has removed all the bricks, the ball could be removed, and the message "Click For Next Level" could appear. Then, when players click, not only would a new ball appear, but a whole new set of bricks.

This page intentionally left blank



6

Picture Puzzles: Sliding and Jigsaw

- Manipulating Bitmap Images
- Sliding Puzzle Game
- Jigsaw Puzzle Game

There is a whole class of games that revolves around using photographs or detailed drawings. Things like jigsaw puzzles are fairly new to computer gaming because it wasn't until the mid-1990s that consumer computers had graphics good enough to display detailed images.

Flash has the ability to import a few different image formats. However, it doesn't stop at just importing them. You can actually get to the bitmap data and manipulate the images. This enables us to cut pictures apart for use in puzzle games.

NOTE

Flash supports JPG, GIF, and PNG file formats. The JPG format is ideal for photographs because it compresses image data well and enables you to determine the amount of compression to use when making the JPG. The GIF format is another compressed format, better suited to drawn graphics of a limited number of colors. The PNG format offers good compression and excellent full-resolution quality. All of these formats can be created by Adobe Fireworks or Adobe Photoshop, which come in some of the Adobe bundles along with Flash.

Let's take a look at the basics behind importing and manipulating images. Then, we'll look at two games that use playing pieces taken from cut-apart imported images.

Manipulating Bitmap Images

SOURCE FILES
http://flashgameu.com
A3GPU06_Bitmap.zip

Before we can play with a bitmap, we must first import it. You could also use a bitmap in the library by just assigning it a class name and then accessing it that way. But, importing an external bitmap is more versatile in almost every way.

Loading a Bitmap

A `Loader` object is a special version of a `Sprite` that pulls its data from an external source. You'll need to pair it up with a `URLRequest`, which handles the network file access.

Here is a very simple class that loads a single JPG file and places it on the screen. After creating a new `Loader` and a new `URLRequest`, we pair them up with the `load` command. The entire process takes only three lines of code. Then, we use `addChild` to add the `Loader` object to the stage, just like a normal display object like a `Sprite`:

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

```
        public class BitmapExample extends MovieClip {

            public function BitmapExample() {
                var loader:Loader = new Loader();
                var request:URLRequest = new URLRequest("myimage.jpg");
                loader.load(request);
                addChild(loader);
            }
        }
}
```

Figure 6.1 shows a bitmap loaded this way. It is positioned in the upper-left corner. Because the `Loader` object acts like a normal display object, we can also set its `x` and `y` position to center it on the screen, or place it anywhere we want.

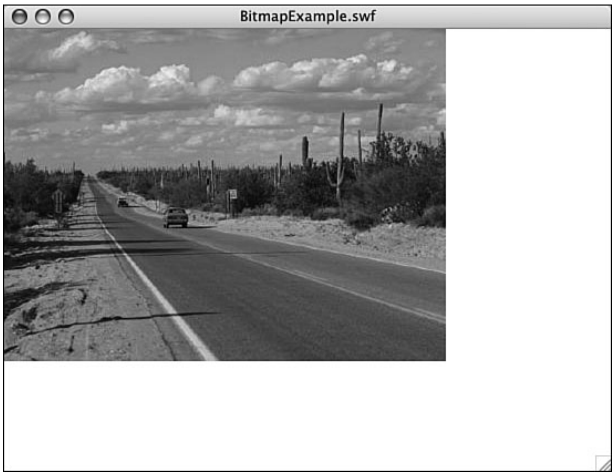


Figure 6.1
This bitmap image was loaded from an external file, but now behaves like a normal display object.

NOTE
Even though `URLRequest` is meant to work from web servers, it works just as well from your local hard drive while testing.

This can prove useful for loading introductory or instruction graphics in a game. For instance, your company logo can appear on the introduction screen using this simple set of commands. Instead of embedding your logo in each Flash game, you can have a single **logo.png** file and use a `Loader` and `URLRequest` to bring it in and display it. Then, one change to **logo.png** and all your games are using the new logo.

Breaking a Bitmap into Pieces

Loading a single bitmap and displaying it is useful, but we need to dig into the data and extract puzzle pieces from the image to build the games in the rest of this chapter.

The first change we need to make to the simple example from before is to recognize when the bitmap is done loading and start processing it. This can be done with an `Event.COMPLETE` listener. We add that to the `Loader` object, and then we can put all of our manipulation code into the `loadingDone` function that it will call.

NOTE

In addition to `Event.COMPLETE`, you can also get status reports of the progress of a downloading image. Look up `URLRequest` in the Flash documentation to see some examples of loading tracking, and even ways to catch and handle errors.

Here is the start of the new class. We'll need the `flash.geom` class later on. I've also put the loading code into its own function, called `loadBitmap`, so that that it will be easy to transport to the games later in this chapter:

```
package {
     import flash.display.*;
     import flash.events.*;
     import flash.net.URLRequest;
     import flash.geom.*;

     public class BitmapExample extends MovieClip {

          public function BitmapExample() {
               loadBitmap("testimage.jpg");
          }

          // get the bitmap from an external source
          public function loadBitmap(bitmapFile:String) {
               var loader:Loader = new Loader();
               loader.contentLoaderInfo.addEventListener(
                              Event.COMPLETE, loadingDone);
               var request:URLRequest = new URLRequest(bitmapFile);
               loader.load(request);
          }
```

When the image is completely loaded, the `loadingDone` function is called. The first thing it does is to create a new `Bitmap` object. It takes the data for this from `event.target.loader.content` (in other words, the `content` property of the original `Loader` object):

```
private function loadingDone(event:Event):void {
    // get loaded data
    var image:Bitmap = Bitmap(event.target.loader.content);
```

We can get the width and height of the bitmap by accessing the `width` and `height` properties of the `image` variable now that it contains the content. What we want to do with these is to get the width and height of each of the puzzle pieces. For this example, we'll have six columns and four rows. So, the total width divided by six gives us the width of each piece. And, the total height divided by four gives use the height of each piece:

```
    // compute the width and height of each piece
    var pieceWidth:Number = image.width/6;
    var pieceHeight:Number = image.height/4;
```

Now we loop through all six columns and four rows to create each puzzle piece:

```
    // loop through all pieces
    for(var x:uint=0;x<6;x++) {
        for (var y:uint=0;y<4;y++) {
```

Creating a puzzle piece is done by first making a new `Bitmap` object. We specify the width and height to create one. Then, we use `copyPixels` to copy a section of the original image into the `bitmapData` property of the puzzle piece.

The `copyPixels` command takes three basic parameters: the image to copy from, the `Rectangle` specifying the part of the image to get, and the `Point` that defines where the piece of the image will go in the new bitmap:

```
            // create new puzzle piece bitmap
            var newPuzzlePieceBitmap:Bitmap =
                        new Bitmap(new BitmapData(pieceWidth,
                        pieceHeight));
            newPuzzlePieceBitmap.bitmapData.copyPixels(
                        image.bitmapData,new Rectangle(x*pieceWidth,
                        y*pieceHeight,pieceWidth,
                        pieceHeight),new Point(0,0));
```

A bitmap by itself isn't our end goal. We want to have a `Sprite` to show on the screen. So, we create a new one, and then add the bitmap to it. Then, we add the `Sprite` to the stage.

```
            // create new sprite and add bitmap data to it
            var newPuzzlePiece:Sprite = new Sprite();
            newPuzzlePiece.addChild(newPuzzlePieceBitmap);

            // add to stage
            addChild(newPuzzlePiece);
```

Finally, we can set the location of the sprites. We want to place them on the screen according to their column and row, plus about five pixels to create some blank spacing between the pieces. We also offset the horizontal and vertical positions of all pieces by 20 pixels. Figure 6.2 shows the 24 puzzle pieces.

```
                // set location
                newPuzzlePiece.x = x*(pieceWidth+5)+20;
                newPuzzlePiece.y = y*(pieceHeight+5)+20;
            }
        }
    }
```

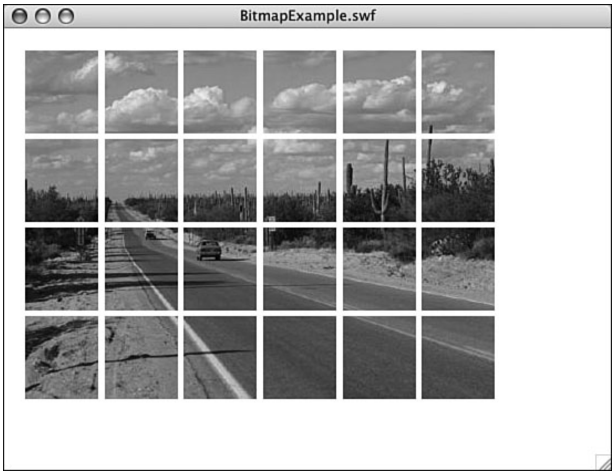


Figure 6.2
The imported image is broken into 24 pieces, which are then spaced apart on the screen.

Now that we know how to create a set of puzzle pieces from an imported image, we can go ahead and build games around them. First, we'll create a simple sliding puzzle. Later, we'll try a more complex jigsaw puzzle game.

Sliding Puzzle Game

SOURCE FILES
http://flashgameu.com
A3GPU06_SlidingPuzzle.zip

It is surprising that a game like the sliding puzzle game had existed way before computers. The physical game was a small handheld plastic square, with usually 15 plastic interlocked pieces inside it. You could slide one of the plastic pieces into the unoccupied 16th slot, and then slide another into the newly unoccupied spot, and so on. This goes on until the puzzle is in order.

The physical version usually didn't involve a picture, but instead the numbers 1 through 15. It was sometimes called the 15-puzzle.

NOTE
The problems with the physical game were that the squares often jammed, frustrating players getting their fingers pinched trying to unstick them. Also, after the puzzle was solved, you needed to look away and randomly move squares for a while to reset it into a random configuration.

As a computer game, this works much better. For one, you can offer different sets of squares like an image. You can offer a new image each time, allowing players to discover the image as the puzzle nears completion. Plus, the computer can randomly arrange the puzzle pieces at the start of each game.

Oh, and no pinching.

In our version of the sliding puzzle game, we use a variable number of pieces cut from the image. In addition, there is a random shuffle of the tiles at the start of the game. Plus, we animate the pieces moving from one spot to the other so that they appear to slide. We also recognize when the solution has been found.

Setting Up the Movie

This game uses the same three-frame framework we have used for the last two chapters: intro, play, and gameover. Instructions are supplied on the first frame.

The only graphic needed is the image itself. This will be an external JPG image called **slidingimage.jpg**. We'll make it 400 pixels by 300 pixels.

NOTE
Picture size and compression are two things to take note of when creating an image for a game like this. All three of the images used in this chapter are less than 34K. Each is 400x300 with 80 percent JPEG compression. It would be easy to produce an image 20 times that in size using lossless compression, like PNG files. But, that level of quality is not needed, and would only result in long download times for the player.

Remember that we'll be cutting the puzzle image up, and then removing the bottom right piece. The player needs this blank spot to make moves. So, it is best to choose an image that doesn't have anything important at the bottom right.

Setting Up the Class

The sliding puzzle game needs the `URLRequest` and `geom` classes to handle the image. We'll also be using a `Timer` object to facilitate the sliding animation:

```
package {
    import flash.display.*;
    import flash.events.*;
    import flash.net.URLRequest;
    import flash.geom.*;
    import flash.utils.Timer;
```

There are plenty of constants to define for this game, starting with the spacing between the pieces and the general offset of all the pieces. We'll also decide how many pieces to cut the image into, in this case 4x3:

```
    public class SlidingPuzzle extends MovieClip {
        // space between pieces and offset
        static const pieceSpace:Number = 2;
        static const horizOffset:Number = 50;
        static const vertOffset:Number = 50;

        // number of pieces
        static const numPiecesHoriz:int = 4;
        static const numPiecesVert:int = 3;
```

NOTE

The number of columns and rows in the puzzle should roughly mirror the dimensions of the image. In this case, we know it is a 400x300 images, and we are making a 4x3 puzzle. So, the pieces will be 100x100 in size. There's nothing wrong with making rectangular pieces, like a 4x4 puzzle with 100x75 pieces. But, you probably don't want to get too far away from square.

To randomize the board at the start of the game, we'll make a number of random moves. We talk more about that later. In the meantime, we need to store the number of random moves in a constant for easy changeability:

```
        // random shuffle steps
        static const numShuffle:int = 200;
```

The puzzle pieces will smoothly slide into place using a `Timer`. We'll decide the number of steps and the length of time it takes for the slider to complete its movement:

```
        // animation steps and time
        static const slideSteps:int = 10;
        static const slideTime:int = 250;
```

The width and height of a puzzle piece will be calculated according to the `numPiecesHoriz` and `numPiecesVert` constants, and the size of the image. We'll get those values just after the image has been loaded:

```
        // size of pieces
        private var pieceWidth:Number;
        private var pieceHeight:Number;
```

We need an array to store the puzzle pieces. We won't be storing just the references to the new sprites here, but a small object that contains the location of the puzzle piece in the finished puzzle as well as the sprite reference:

```
        // game pieces
        private var puzzleObjects:Array;
```

We need a host of variables to track game play and movement. First, we have `blankPoint`, which is a `Point` object indicating the location of the blank spot in the puzzle. When the player clicks a piece adjacent to the blank spot, the piece slides over into it. The `slidingPiece` holds a reference to the piece moving, and the `slideDirection` and `slideAnimation Timer` will facilitate this animation:

```
        // tracking moves
        private var blankPoint:Point;
        private var slidingPiece:Object;
        private var slideDirection:Point;
        private var slideAnimation:Timer;
```

When players press the Start button, they will go to the second frame, which calls `startSlidingPuzzle`. Unlike constructor functions in other games, this one doesn't do much. This is because until the image is loaded, there is not much to do.

The `blankPoint` variable is set to the bottom left, using some of the constants. Then, `loadBitmap` is called with the name of the image file:

```
public function startSlidingPuzzle() {
    // blank spot is the bottom right
    blankPoint = new Point(numPiecesHoriz-1,numPiecesVert-1);

    // load the bitmap
    loadBitmap("slidingpuzzle.jpg");
}
```

NOTE

Remember that we begin counting in arrays and loops in ActionScript with zero. So, the piece at the upper left is 0,0. Thus, the piece at the lower right is one less than the number of pieces wide, or `numPiecesHoriz-1`. So, if a puzzle were four pieces across and three down, the piece at the lower right would be 3,2 or `numPiecesHoriz-1,numPiecesVert-1`.

Loading the Image

The `loadBitmap` function is identical to the one used in the example earlier in this chapter:

```
// get the bitmap from an external source
public function loadBitmap(bitmapFile:String) {
     var loader:Loader = new Loader();
     loader.contentLoaderInfo.addEventListener(Event.COMPLETE, loadingDone);
     var request:URLRequest = new URLRequest(bitmapFile);
     loader.load(request);
}
```

The `loadingDone` function becomes much more important in this case than the earlier example. Now that the image has been loaded, the width and height can be obtained, which will give us the individual width and height of each piece. We can set those variables, and then call `makePuzzlePieces` to do the cutting. Finally, `shufflePuzzlePieces` will randomize the puzzle and get it ready for the player:

```
// bitmap done loading, cut into pieces
public function loadingDone(event:Event):void {
     // create new image to hold loaded bitmap
     var image:Bitmap = Bitmap(event.target.loader.content);
     pieceWidth = image.width/numPiecesHoriz;
     pieceHeight = image.height/numPiecesVert;

     // cut into puzzle pieces
     makePuzzlePieces(image.bitmapData);

     // shuffle them
     shufflePuzzlePieces();
}
```

Cutting the Bitmap into Pieces

Although our earlier example cut the image into pieces, it didn't have to build all the data objects needed to make them useful in a game. The function `makePuzzlePieces` does this by creating the array `puzzleObjects`. After the puzzle piece sprite is created, and the position of the sprite set, the temporary variable `newPuzzleObject` is created.

In `newPuzzleObject`, three properties are attached. The first is `currentLoc`, which is a `Point` object that shows where the puzzle piece is currently. For instance, 0,0 would mean the upper left; 3,2 would mean the lower right.

Similarly, `homeLoc` contains a `Point`, too. However, this is the original (and final) location for the piece. It will not change during the game, and provides a point of reference so that we can determine when each piece has returned to its correct position.

NOTE

Another way to go here would be to store the `currentLoc` and `homeLoc` as properties of the sprites. Then, store only the sprites in the array. In the first case, the three values would be `puzzleObjects[x].currentLoc`, `puzzleObjects[x].homeLoc`, and `puzzleObjects[x].piece`. In the latter case, the same data would be `puzzleObjects[x].currentLoc`, `puzzleObjects[x].homeLoc`, and `puzzleObjects[x]` (without the `.piece` because the item in the array *is* the sprite). I prefer creating my own array of objects to ensure that ActionScript can quickly get the information without having to look at the entire `Sprite` object each time.

We also have a `piece` property of `newPuzzleObject`. The `piece` property holds a reference to the piece's sprite.

We'll store all the `newPuzzleObject` variables we create in the `puzzleObjects` array:

```
// cut bitmap into pieces
public function makePuzzlePieces(bitmapData:BitmapData) {
    puzzleObjects = new Array();
    for(var x:uint=0;x<numPiecesHoriz;x++) {
        for (var y:uint=0;y<numPiecesVert;y++) {
            // skip blank spot
            if (blankPoint.equals(new Point(x,y))) continue;

            // create new puzzle piece bitmap and sprite
            var newPuzzlePieceBitmap:Bitmap =
                        new Bitmap(new BitmapData(pieceWidth,pieceHeight));
            newPuzzlePieceBitmap.bitmapData.copyPixels(bitmapData,
                        new Rectangle(x*pieceWidth,y*pieceHeight,
                        pieceWidth,pieceHeight),new Point(0,0));
            var newPuzzlePiece:Sprite = new Sprite();
            newPuzzlePiece.addChild(newPuzzlePieceBitmap);
            addChild(newPuzzlePiece);

            // set location
            newPuzzlePiece.x = x*(pieceWidth+pieceSpace) + horizOffset;
            newPuzzlePiece.y = y*(pieceHeight+pieceSpace) + vertOffset;

            // create object to store in array
            var newPuzzleObject:Object = new Object();
            newPuzzleObject.currentLoc = new Point(x,y);
            newPuzzleObject.homeLoc = new Point(x,y);
            newPuzzleObject.piece = newPuzzlePiece;
            newPuzzlePiece.addEventListener(MouseEvent.CLICK,
                        clickPuzzlePiece);
            puzzleObjects.push(newPuzzleObject);
            }
        }
    }
```

Each puzzle piece gets its own event listener to listen for mouse clicks. It will call `clickPuzzlePiece`.

At this point, the pieces are all in place, but not shuffled. If we didn't shuffle them at all, the game would start out looking like Figure 6.3.

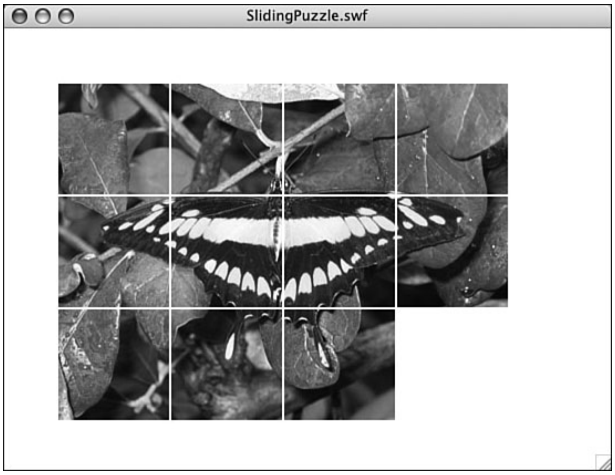


Figure 6.3
The sliding puzzle without shuffling. The piece at the bottom right has been removed to create a space to slide pieces into.

Shuffling the Pieces

After the puzzle pieces are there, we need to shuffle them. The idea is to "mess the puzzle up" so that the player has the challenge of putting it back in order.

One way to mess the puzzle up is to just place all the pieces at random locations. However, that isn't the right way to do it. There is a good chance that if you simply place the pieces randomly, you will end up with an arrangement that can never be reordered properly. Figure 6.4 demonstrates one such situation.

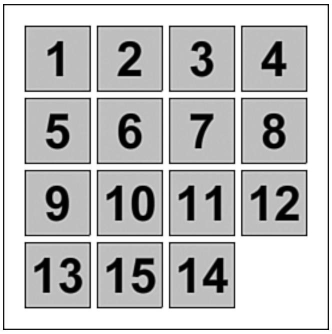


Figure 6.4
With simply the 15th and 14th piece reversed, this puzzle cannot be solved.

So, instead of randomly placing each piece, we'll start with the complete puzzle, and then make random moves until the board looks completely random.

The shufflePuzzlePieces function loops and calls shuffleRandom a number of times. It is shuffleRandom that does the real work here:

```
// make a number of random moves
public function shufflePuzzlePieces() {
     for(var i:int=0;i<numShuffle;i++) {
          shuffleRandom();
     }
}
```

To make a random move, we'll look at all the pieces on the board. Then, we'll determine what moves are possible and put them in an array. We then pick a random move from that array and do it.

The key to this will be the validMove function, which we examine next. The shuffleRandom function, after it has picked a random move, calls movePiece, which is the same function we use when the player clicks to make a move:

```
// random move
public function shuffleRandom() {
     // loop to find valid moves
     var validPuzzleObjects:Array = new Array();
     for(var i:uint=0;i<puzzleObjects.length;i++) {
          if (validMove(puzzleObjects[i]) != "none") {
               validPuzzleObjects.push(puzzleObjects[i]);
          }
     }
     // pick a random move
     var pick:uint = Math.floor(Math.random()*validPuzzleObjects.length);
     movePiece(validPuzzleObjects[pick],false);
}
```

The validMove function takes as a parameter a reference to a puzzleObject. Using the currentLoc property of this puzzle piece, it can determine whether the piece is next to the blank spot.

First, it looks up, above the puzzle piece. In this case, the x values of both the piece and the blankSpot should match. If they do, the vertical, or y, positions are compared. The blankPoint.y should be one less than the currentLoc.y of the puzzleObject. If this all works out, "up" is returned, which tells the function calling validMove that this piece does indeed have a valid move: "up".

NOTE

Notice that the `validMove` function declaration is for it to return a string. You can see the `": String"` on the first line of code that follows. It is always a good idea to indicate what type of data will be returned by the function. You'll be helping the Flash player perform more efficiently when you do.

Then, the down, left, and right moves are explored. If none of these turns up a valid move, the value `"none"` is returned. This means there is no valid move for this puzzle piece:

```
public function validMove(puzzleObject:Object): String {
     // is the blank spot above
     if ((puzzleObject.currentLoc.x == blankPoint.x) &&
          (puzzleObject.currentLoc.y == blankPoint.y+1)) {
          return "up";
     }
     // is the blank spot below
     if ((puzzleObject.currentLoc.x == blankPoint.x) &&
          (puzzleObject.currentLoc.y == blankPoint.y-1)) {
          return "down";
     }
     // is the blank to the left
     if ((puzzleObject.currentLoc.y == blankPoint.y) &&
          (puzzleObject.currentLoc.x == blankPoint.x+1)) {
          return "left";
     }
     // is the blank to the right
     if ((puzzleObject.currentLoc.y == blankPoint.y) &&
          (puzzleObject.currentLoc.x == blankPoint.x-1)) {
          return "right";
     }
     // no valid moves
     return "none";
}
```

After a shuffle is done, the game starts with the pieces all mixed up, as shown in Figure 6.5.

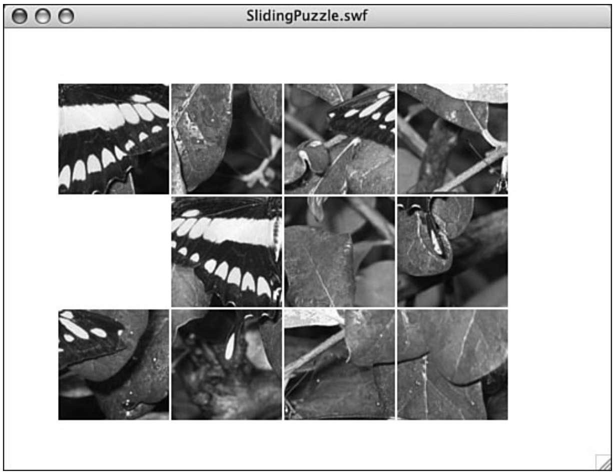


Figure 6.5
The game now starts with the pieces shuffled.

NOTE
The issue of how many times to shuffle the pieces doesn't have a definitive solution. I choose 200 because it seems about right. If you choose too few, the solution will be easier. Too many, and you'll start to see a pause at the start of the game while the shuffle takes place.

Reacting to Player Clicks

When the player clicks, the `clickPuzzlePiece` function will run. The `event` passed to it will have a `currentTarget` that will match a `piece` in the `puzzleObjects` list. A quick loop will find out which one matches, and then the `movePiece` function is called:

```
// puzzle piece clicked
public function clickPuzzlePiece(event:MouseEvent) {
     // find piece clicked and move it
     for(var i:int=0;i<puzzleObjects.length;i++) {
          if (puzzleObjects[i].piece == event.currentTarget) {
               movePiece(puzzleObjects[i],true);
               break;
          }
     }
}
```

Notice that when the `shuffleRandom` function called `movePiece`, it used `false` as the second parameter. When `clickPuzzlePiece` called `movePiece`, however, it used `true` as the second parameter.

The second parameter is `slideEffect`, and it takes either `true` or `false` as a value. If `true`, a `Timer` object is created to move the piece gradually over a short period of time. If `false`, the piece immediately moves. We wanted the pieces to move immediately in the case of the shuffle, but when the player is making moves, we want to use the animation.

The decision is not actually completed inside of `movePiece`. All `movePiece` does is to call `validMove` and to determine whether the move is up, down, left, or right. Then, it calls `movePieceInDirection` with the same `puzzleObject` and `slideEffect` values, as well as new `dx` and `dy` values according to the direction of movement.

NOTE

The `movePiece` function uses the `switch` structure in ActionScript to branch into one of four pieces of code. The `switch` is like a series of `if…then` statements, but the tested variable is only needed in the `switch` line. Each branch starts with `case` and the value that needs to be matched. Each branch must end with a `break` command.

```
// move a piece into the blank space
public function movePiece(puzzleObject:Object, slideEffect:Boolean) {
     // get direction of blank space
     switch (validMove(puzzleObject)) {
          case "up":
               movePieceInDirection(puzzleObject,0,-1,slideEffect);
               break;
          case "down":
               movePieceInDirection(puzzleObject,0,1,slideEffect);
               break;
          case "left":
               movePieceInDirection(puzzleObject,-1,0,slideEffect);
               break;
          case "right":
               movePieceInDirection(puzzleObject,1,0,slideEffect);
               break;
     }
}
```

The `movePieceInDirection` function changes the `currentLoc` of the piece and the `blankPoint` variable instantly. This sets the game up for handling the situation where the player makes another move before the animation is done. The piece and the `blankPoint` are already really in the correct location. The animation is just cosmetic:

```
// move the piece into the blank spot
public function movePieceInDirection(puzzleObject:Object,
                    dx,dy:int, slideEffect:Boolean) {
```

```
        puzzleObject.currentLoc.x += dx;
        puzzleObject.currentLoc.y += dy;
        blankPoint.x -= dx;
        blankPoint.y -= dy;
```

If there is to be an animation, the `startSlide` function is called to set it up. Otherwise, the puzzle piece is moved immediately to the new location:

```
        // animate or not
        if (slideEffect) {
            // start animation
            startSlide(puzzleObject,dx*(pieceWidth+pieceSpace),
                    dy*(pieceHeight+pieceSpace));
        } else {
            // no animation, just move
            puzzleObject.piece.x =
                    puzzleObject.currentLoc.x*(pieceWidth+pieceSpace) + horizOffset;
            puzzleObject.piece.y =
                    puzzleObject.currentLoc.y*(pieceHeight+pieceSpace) + vertOffset;
        }
    }
```

Animating the Slide

The animated slide is done by starting a `Timer` object and moving the puzzle piece in steps. According to the constants at the start of the class, there should be 10 steps, all 250 milliseconds apart.

The `startSlide` function first sets up some variables to track the animation. The puzzle piece to be moved is `slidingPiece`. The `slideDirection` is a `Point` object with `dx` and `dy` as the movement direction. It will be either 1,0, –1,0, 0,1 or 0,–1 depending on which of the four directions the piece is going.

Then, the `Timer` is created, and two listeners attached to it. The `TimerEvent.TIMER` listener moves the piece, whereas the `TimerEvent.TIMER_COMPLETE` listener calls `slideDone` to wrap the animation up:

```
    // set up a slide
    public function startSlide(puzzleObject:Object, dx, dy:Number) {
        if (slideAnimation != null) slideDone(null);
        slidingPiece = puzzleObject;
        slideDirection = new Point(dx,dy);
        slideAnimation = new Timer(slideTime/slideSteps,slideSteps);
        slideAnimation.addEventListener(TimerEvent.TIMER,slidePiece);
        slideAnimation.addEventListener(TimerEvent.TIMER_COMPLETE,slideDone);
        slideAnimation.start();
    }
```

Every 250 milliseconds, the puzzle piece moves a fraction of the distance closer to its final destination.

Also notice the first line of `startSlide` is a potential call to `slideDone`. This is done if a new slide animation is about to start but the previous one has yet to finish. If so, the previous one is quickly ended with a single call to `slideDone` that places the animating piece in its final location, thus clearing the way for a new slide animation.

NOTE

This is a different type of animation from the time-based animation used in Chapter 5, "Game Animation: Shooting and Bouncing Games." The animated slide here is not a critical game element, just a cosmetic one. So, it makes sense to place it outside of the rest of the game logic with its own temporary timer. We don't need to worry about performance remaining consistent because it won't affect game play.

```
// move one step in slide
public function slidePiece(event:Event) {
    slidingPiece.piece.x += slideDirection.x/slideSteps;
    slidingPiece.piece.y += slideDirection.y/slideSteps;
}
```

When the `Timer` is done, the location of the puzzle piece will be set to be sure the piece is exactly where it should be. Then, the `slideAnimation` timer is removed.

This is also time to call `puzzleComplete` to see whether every piece is in the right place. If so, `clearPuzzle` is called, and the main timeline goes to the gameover frame:

```
// complete slide
public function slideDone(event:Event) {
    slidingPiece.piece.x =
         slidingPiece.currentLoc.x*(pieceWidth+pieceSpace) + horizOffset;
    slidingPiece.piece.y =
         slidingPiece.currentLoc.y*(pieceHeight+pieceSpace) + vertOffset;
    slideAnimation.stop();
        slideAnimation = null;

    // check to see if puzzle is complete now
    if (puzzleComplete()) {
        clearPuzzle();
        gotoAndStop("gameover");
    }
}
```

Game Over and Cleanup

Determining whether the game is over is as easy as looking at each piece and comparing the `currentLoc` with the `homeLoc`. Thanks to the `equals` function of the `Point` class, we can do this in one step.

If all the pieces are in their original locations, `true` is returned:

```
// check to see if all pieces are in place
public function puzzleComplete():Boolean {
     for(var i:int=0;i<puzzleObjects.length;i++) {
          if (!puzzleObjects[i].currentLoc.equals(puzzleObjects[i].homeLoc)) {
               return false;
          }
     }
     return true;
}
```

Here is the cleanup function for the game. All the puzzle piece sprites get rid of their `MouseEvent.CLICK` events and are removed, and then the `puzzleObjects` array is set to `null`. Because the puzzles pieces were the only game objects created, that's all that is needed:

```
// remove all puzzle pieces
public function clearPuzzle() {
     for (var i in puzzleObjects) {
           puzzleObjects[i].piece.addEventListener(MouseEvent.CLICK,
                     clickPuzzlePiece);
          removeChild(puzzleObjects[i].piece);
     }
     puzzleObjects = null;
}
```

Modifying the Game

The game play of this game is pretty straightforward and probably won't benefit from any variation. However, you can improve the program itself by allowing ways for the image to be dynamically selected. For instance, the web page could pass in the name of the image. Then, you could have one game that called on different images depending on the web page, or perhaps the day.

You could also make this game progressive. After each puzzle is complete, a new level is presented. The image filename and the `numPiecesHoriz` and `numPiecesVert` could be passed in as parameters to `startSlidingPuzzle`. So, instead of going to gameover when the puzzle is complete, it goes to the next level with a larger, harder image with more pieces.

Jigsaw Puzzle Game

SOURCE FILES
http://flashgameu.com
A3GPU06_JigsawPuzzle.zip

Jigsaw puzzles first became popular in the 18th century when they were made from wood using an actual jigsaw. Today, they are made from cardboard with a cutting press. Today's puzzles can include as many as 24,000 pieces.

Computer jigsaw puzzles have been popping up around the Web and in casual game CD collections since the late 90s. In this chapter, we build a simple jigsaw puzzle using rectangular pieces cut from an imported image.

NOTE
Most jigsaw puzzle games go the whole way and make the puzzle pieces look like traditional puzzle pieces with interlocking tabs. This cosmetic feature can be done in Flash, but only by using quite a bit of vector drawing and bitmap-manipulation tools. To stick with the fundamentals, we use rectangle-shaped pieces here.

In our jigsaw puzzle game, we'll cut out pieces just like the sliding puzzle game. Instead of placing them in particular spots on the screen, however, we throw them all around randomly. Then, the player can drag pieces around the screen.

The difficult part of the coding is making the pieces lock together when the player puts them side by side.

Setting Up the Class

The jigsaw puzzle movie is pretty much identical to the sliding puzzle. There are three frames, and the second calls `startJigsawPuzzle`. The same `imports` are needed, too:

```
package {
     import flash.display.*;
     import flash.events.*;
     import flash.net.URLRequest;
     import flash.geom.*;
     import flash.utils.Timer;
```

The `numPiecesHoriz` and `numPiecesVert` are listed right at the top of the class. Changes to these constants are very likely because you can really vary the number of pieces in a jigsaw puzzle to create different levels of difficulty. We'll create an 8x6 jigsaw puzzle for this example:

```
    public class JigsawPuzzle extends MovieClip {
        // number of pieces
        const numPiecesHoriz:int = 8;
        const numPiecesVert:int = 6;
```

The width and height of the pieces will be decided after the image has been imported and the program can tell the size of the image:

```
        // size of pieces
        var pieceWidth:Number;
        var pieceHeight:Number;
```

Like with the sliding puzzle, we store all the puzzle pieces as objects in the `puzzleObjects` array:

```
        // game pieces
        var puzzleObjects:Array;
```

Here is where the jigsaw puzzle game really starts to deviate from the sliding puzzle. Instead of putting the puzzle pieces directly on the stage, we place them in one of two sprites. The `selectedPieces` sprite will hold any puzzle pieces that are currently being dragged. The `otherPieces` sprite will hold everything else.

NOTE

Placing groups of display objects inside of sprites is a great way to group similar objects together. As you'll see later in this example, you can use `addChild` to move a display object from one sprite to another.

```
        // two levels of sprites
        var selectedPieces:Sprite;
        var otherPieces:Sprite;
```

When players select a piece to drag, they could be selecting a single piece, or they could be selecting a group of linked pieces. So, instead of a single variable pointing to the piece being dragged, we need an array to store one or more pieces:

```
        // pieces being dragged
        var beingDragged:Array  = new Array();
```

The game constructor function, in addition to calling `loadBitmap`, also creates the two sprites we need and adds them to the stage. The order in which these are added is important because we want the `selectedPieces` sprite to be on top of the `otherPieces` sprite:

```
// load picture and set up sprites
public function startJigsawPuzzle() {
    // load the bitmap
```

```
        loadBitmap("jigsawimage.jpg");

        // set up two sprites
        otherPieces = new Sprite();
        selectedPieces = new Sprite();
        addChild(otherPieces);
        addChild(selectedPieces); // selected on top
    }
```

Loading and Cutting the Image

The image is loaded in the same way as the sliding puzzle. I'll skip the `loadBitmap` function because it is identical to the previous one.

Loading the Bitmap Image

The `loadingDone` function isn't too far off either. After the image is brought in and `pieceWidth` and `pieceHeight` are calculated, `makePuzzlePieces` is called to cut the image up.

We're doing something a little different here with the `pieceWidth` and `pieceHeight` calculations. The `Math.floor` function is being used, along with the division by ten to restrict the width and height numbers to multiples of ten. So, for instance, if we were doing 7 pieces across, into 400 pixels wide, we'd have 57.14 pixels across for each piece. But we'll be using a 10x10 grid here to allow players to match up pieces side by side easier. So, by rounding that down to 50, we ensure that the widths and heights will match up on a 10x10 grid. More about this when we build the `lockPieceToGrid` function later on.

Finally, two event listeners are added. The first is an `ENTER_FRAME` event for piece movement during dragging. The second is a `MOUSE_UP` event on the stage. This is needed to get mouse up events that signal the end of dragging.

The user clicks a piece to start a drag, and the `MOUSE_DOWN` event acts on the piece itself. But then when the drag is complete, we can't rely on the fact that the mouse will still be over the same, or any, piece. The player may be moving the cursor quickly and have the cursor slightly off any pieces. However, mouse events carry through to the stage, so we are safer placing a `MOUSE_UP` listener on the stage to make sure we get notified.

```
    // bitmap done loading, cut into pieces
    private function loadingDone(event:Event):void {
        // create new image to hold loaded bitmap
        var image:Bitmap = Bitmap(event.target.loader.content);
        pieceWidth = Math.floor((image.width/numPiecesHoriz)/10)*10;
        pieceHeight = Math.floor((image.height/numPiecesVert)/10)*10;

        // place loaded bitmap in image
        var bitmapData:BitmapData = image.bitmapData;

        // cut into puzzle pieces
        makePuzzlePieces(bitmapData);

        // set up movement and mouse up events
        addEventListener(Event.ENTER_FRAME,movePieces);
        stage.addEventListener(MouseEvent.MOUSE_UP,liftMouseUp);
    }
```

Cutting the Puzzle Pieces

The basics for cutting the pieces are the same in this game. However, we don't need to set any location for the pieces because they will be arranged randomly a bit later.

After the sprites are created, they are added to `otherPieces`, which is the bottom of the two sprites we created.

The `puzzleObject` elements are also slightly different. Instead of `currentLoc` and `homeLoc`, we just have `loc`, which is a `Point` object that tells us where the puzzle piece belongs in the complete puzzle. 0,0 for instance, is the top left piece.

Also, we're added a `dragOffset` property to the puzzle pieces. We use this to position each piece the proper distance from the cursor while dragging:

```
// cut bitmap into pieces
private function makePuzzlePieces(bitmapData:BitmapData) {
    puzzleObjects = new Array();
    for(var x:uint=0;x<numPiecesHoriz;x++) {
        for (var y:uint=0;y<numPiecesVert;y++) {
            // create new puzzle piece bitmap and sprite
            var newPuzzlePieceBitmap:Bitmap =
                        new Bitmap(new BitmapData(pieceWidth,pieceHeight));
            newPuzzlePieceBitmap.bitmapData.copyPixels(bitmapData,
                        new Rectangle(x*pieceWidth,y*pieceHeight,
                        pieceWidth,pieceHeight),new Point(0,0));
            var newPuzzlePiece:Sprite = new Sprite();
            newPuzzlePiece.addChild(newPuzzlePieceBitmap);

            // place in bottom sprite
            otherPieces.addChild(newPuzzlePiece);

            // create object to store in array
            var newPuzzleObject:Object = new Object();
            newPuzzleObject.loc = new Point(x,y); // location in puzzle
            newPuzzleObject.dragOffset = null; // offset from cursor
            newPuzzleObject.piece = newPuzzlePiece;
            newPuzzlePiece.addEventListener(MouseEvent.MOUSE_DOWN,
                              clickPuzzlePiece);
                    puzzleObjects.push(newPuzzleObject);
               }
          }

          // randomize locations of pieces
          shufflePieces();
     }
```

The shuffle function picks random locations for all the puzzle pieces. We don't care whether the pieces fall on top of each other, or how they are distributed. It should be as if they all just fell out of the box. Figure 6.6 shows such a random distribution.

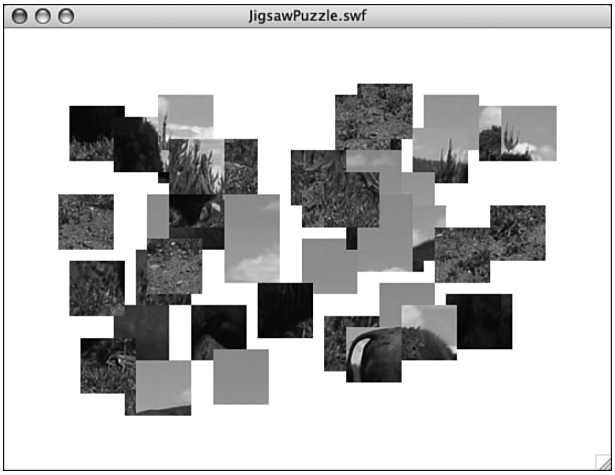


Figure 6.6
The pieces of the jigsaw puzzle are randomly arranged on the screen.

```
// random locations for the pieces
public function shufflePieces() {
     // pick random x and y
     for(var i in puzzleObjects) {
          puzzleObjects[i].piece.x = Math.random()*400+50;
          puzzleObjects[i].piece.y = Math.random()*250+50;
     }
     // lock all pieces to 10x10 grid
     lockPiecesToGrid();
}
```

The last line of `shufflePieces` calls `lockPieceToGrid`. This function loops through all the puzzle pieces and moves them to the closest location to a 10x10 grid. For instance, if a piece is at 43,87 it will be moved to 40,90.

The reason for using `lockPieceToGrid` is that it is a simple way to allow the player to move one piece close, but not exactly next to, another piece and have the pieces lock together. Normally, if a piece is just one pixel away from another, it won't lock. By putting all the pieces on a 10x10 grid, however, it means that pieces are either perfectly matching, or they are ten pixels away:

```
// take all pieces and lock them to the nearest 10x10 location
public function lockPiecesToGrid() {
     for(var i in puzzleObjects) {
          puzzleObjects[i].piece.x =
                    10*Math.round(puzzleObjects[i].piece.x/10);
          puzzleObjects[i].piece.y =
                    10*Math.round(puzzleObjects[i].piece.y/10);
     }
}
```

NOTE
Because we are using puzzle pieces that are multiples of ten wide and high, it is better to use a source image that has dimensions that are multiples of ten. For instance a 400x300 image will be used completely. However, a 284x192 image will lose a bit from the right and bottom in an effort to restrict piece sizes to multiples of ten.

Dragging Puzzle Pieces

When a player clicks a puzzle piece, several things need to happen before the piece can move along with the cursor. The first is to figure out which piece was clicked.

Determining which Piece Was Clicked

This is done by looping through the `puzzleObjects` until one is found where the `piece` property matches the `event.currentTarget`.

Then, this piece is added to an empty `beingDragged` array. In addition, the `dragOffset` for this piece is calculated by the distance between the click location and the sprite.

The sprite is moved from the bottom `otherPieces` sprite to the top `selectedPieces` sprite. It just takes one call to `addChild` to do that. This means that as the player drags the puzzle piece, it floats above all other pieces that are left behind on the `otherPieces` sprite:

```
public function clickPuzzlePiece(event:MouseEvent) {
     // click location
     var clickLoc:Point = new Point(event.stageX, event.stageY);

     beingDragged = new Array();

     // find piece clicked
     for(var i in puzzleObjects) {
          if (puzzleObjects[i].piece == event.currentTarget) { // this is it
               // add to drag list
```

```
                beingDragged.push(puzzleObjects[i]);
                // get offset from cursor
                puzzleObjects[i].dragOffset = new Point(clickLoc.x -
                            puzzleObjects[i].piece.x, clickLoc.y -
                            puzzleObjects[i].piece.y);
                // move from bottom sprite to top one
                selectedPieces.addChild(puzzleObjects[i].piece);
                // find other pieces locked to this one
                findLockedPieces(i,clickLoc);
                break;
            }
        }
    }
```

Finding Linked Pieces

The most interesting thing that goes on when the player clicks a piece is the call to `findLockedPieces`. After all, this piece may not be alone. It might have already joined with other puzzle pieces in previous moves. Any pieces joined with the one clicked need to also be added to the `beingDragged` list.

The way to determine whether a piece is locked to another is through a series of steps. This process begins by creating a sorted list of all the pieces, other than the one clicked. This list is sorted by the distance from the original clicked piece.

NOTE

The `sortOn` command is a powerful way to sort lists of objects. Assuming the array contains only similar objects, all with the same sorting property, you can quickly and easily sort the list. For instance, the array `[{a: 4, b: 7}, {a: 3, b:12}, {a: 9, b: 17}]` can be sorted on a by simply programming `myArray.sortOn("a");`.

In the following code, an array is created with the `dist` property and `num` property for each element in the array. The first is the distance of the piece from the original clicked piece. The second is the number of the piece as it appears in `puzzleObjects`:

```
// find pieces that should move together
public function findLockedPieces(clickedPiece:uint, clickLoc:Point) {
    // get list of puzzle objects sorted by distance to the clicked object
    var sortedObjects:Array = new Array();
    for (var i in puzzleObjects) {
        if (i == clickedPiece) continue;
        sortedObjects.push(
                {dist: Point.distance(puzzleObjects[clickedPiece].loc,
                puzzleObjects[i].loc), num: i});
    }
    sortedObjects.sortOn("dist" ,Array.DESCENDING);
```

Now that we have a sorted array of pieces, we can loop through that and check out each piece to see whether it is linked to the original selection.

First, we check the `x` and `y` position of the piece. What we are looking for is to see whether the piece is positioned correctly relative to the original selection. For instance, if the pieces are 50 wide and 50 high, and the original piece is at 170,240 the piece directly to the left of it should be at 120,240. The piece two away to the left should be 70,240 and so on.

Figure 6.7 shows a few examples of piece that do and do not connect.

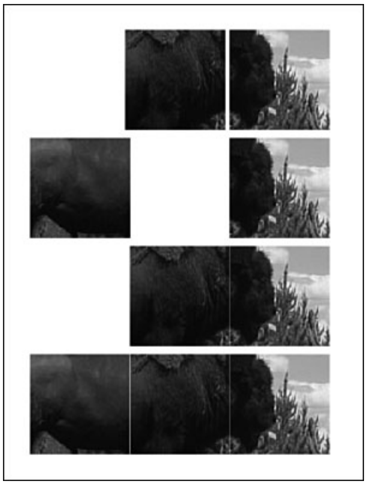


Figure 6.7
The top two examples do not connect, but the bottom two examples do.

In Figure 6.7, the top example shows two pieces that are positioned close, but not close enough to connect.

The next example shows pieces positioned perfectly, but missing a connecting piece between them. The piece on the left is the exact right distance from the piece on the right, but the pieces shouldn't connect because they aren't next to each other and the piece that would connect them is missing

The third example shows two pieces connected. They are adjacent, and the perfect distance. The fourth example is the same, with all three connected.

So, the first step is to see whether the piece under examination is correctly placed. The next step is to see whether it is connected to the pieces found so far. That is delegated to the `isConnected` function, which you will see in a moment.

If the piece is indeed connected, we can add it to the `beingDragged` list, set its `dragOffset` property, and add it to the `selectedPieces` sprite:

We can put this all in a `do` loop so we can repeat it multiple times. It is possible to have a set of linked pieces in a U shape, and try to grab one end of the U. This would mean that the other end of the U would not be recognized as a linked piece. But if we loop through the unlinked pieces again, it will find the U. So, we can set a Boolean variable `oneLineFound` to `true` when we find a link. If we go through all the unlinked pieces without finding a link, then we must have gotten them all. Otherwise, we'll keep looping.

```
    do {
        var oneLinkFound:Boolean = false;
        // look at each object, starting with closest
        for(i=sortedObjects.length-1;i>=0;i--) {
            var n:uint = sortedObjects[i].num; // actual object number
            // get the position relative to the clicked object
            var diffX:int = puzzleObjects[n].loc.x -
                         puzzleObjects[clickedPiece].loc.x;
            var diffY:int = puzzleObjects[n].loc.y -
                         puzzleObjects[clickedPiece].loc.y;
            // see if this object is appropriately placed
                // to be locked to the clicked one
            if (puzzleObjects[n].piece.x ==
                              (puzzleObjects[clickedPiece].piece.x +
                              pieceWidth*diffX)) {
                if (puzzleObjects[n].piece.y ==
                                      (puzzleObjects[clickedPiece].piece.y +
                                      pieceHeight*diffY)) {
                    // see if this object is adjacent to one already selected
                    if (isConnected(puzzleObjects[n])) {
                        // add to selection list and set offset
                        beingDragged.push(puzzleObjects[n]);
                        puzzleObjects[n].dragOffset =
                                            new Point(clickLoc.x -
                                            puzzleObjects[n].piece.x,
                                      clickLoc.y-
                                            puzzleObjects[n].piece.y);
                        // move to top sprite
                        selectedPieces.addChild(
                                            puzzleObjects[n].piece);
                        // link found, remove from array
                        oneLinkFound = true;
                        sortedObjects.splice(i,1);
                    }
                }
            }
        }
    } while (oneLinkFound);
}
```

The key to using the `isConnected` function is the fact that we already sorted the list of pieces by distance to the first piece. This is important because we should be moving outward in the search for new connected pieces. As we examine each new piece for connectivity, pieces between it and the original selection will most likely already have been examined and added to the `beingDragged` array. This minimizes the number of times we need to loop through the array of pieces.

For instance, if the clicked piece is 2,0 and we look at 0,0 next, we can determine it is not connected because 1,0 isn't in `beingDragged`. Then, we look at 1,0 and find that it is connected, but it is too late to add 0,0 because we already looked at it. So, we need to look at 1,0 first because it is closer, and then 0,0, which is farther.

NOTE

If you think this process is complicated, consider that it would usually be done using a computer science process known as recursion. *Recursion* is when a function calls itself. It also causes many freshmen computer science students to switch majors to business. So, I've come up with the methods in this chapter to specifically avoid recursion.

Determining If Pieces Are Connected

The `isConnected` function gets the difference between the horizontal and vertical position of the piece under examination and each piece already in `beingDragged`. If it finds that it is only one away horizontally or vertically (but not both), it is indeed connected:

```
// takes an object and determines if it is directly next to one already selected
public function isConnected(newPuzzleObject:Object):Boolean {
    for(var i in beingDragged) {
        var horizDist:int =
                  Math.abs(newPuzzleObject.loc.x - beingDragged[i].loc.x);
        var vertDist:int =
                  Math.abs(newPuzzleObject.loc.y - beingDragged[i].loc.y);
        if ((horizDist == 1) && (vertDist == 0)) return true;
        if ((horizDist == 0) && (vertDist == 1)) return true;
    }
    return false;
}
```

Moving the Pieces

Finally, we know which pieces should be dragged. They are now all neatly in `beingDragged`, which is used by `movePieces` in every frame to reposition all of them:

```
// move all selected pieces according to mouse location
public function movePieces(event:Event) {
    for (var i in beingDragged) {
        beingDragged[i].piece.x = mouseX - beingDragged[i].dragOffset.x;
        beingDragged[i].piece.y = mouseY - beingDragged[i].dragOffset.y;
    }
}
```

Ending Movement

When the player releases the mouse, the dragging ends. We need to move all the pieces from the `selectedPieces` sprite back down to the `otherPieces` sprite. We'll also call `lockPiecesToGrid` to make sure they can be matched up with the pieces that were not dragged.

NOTE

When the `addChild` function is used to move the pieces back down to the `otherPieces` sprite, they are added above all the other pieces there. This works out nicely because they were already floating above them. The result is that they maintain the appearance of being on top.

```
// stage sends mouse up event, drag is over
public function liftMouseUp(event:MouseEvent) {
    // lock all pieces back to grid
    lockPiecesToGrid();
    // move pieces back to bottom sprite
    for(var i in beingDragged) {
        otherPieces.addChild(beingDragged[i].piece);
    }
    // clear drag array
    beingDragged = new Array();

    // see if the game is over
    if (puzzleTogether()) {
        cleanUpJigsaw();
        gotoAndStop("gameover");
    }
}
```

Game Over

When the mouse is released, we also want to check to see whether the game is over. To do this, we can loop through all the puzzle pieces and compare their positions to the position of the first piece at the upper left. If they are all in the exact correct position relative to that one, we know the puzzle is done:

```
public function puzzleTogether():Boolean {
    for(var i:uint=1;i<puzzleObjects.length;i++) {
        // get the position relative to the first object
        var diffX:int = puzzleObjects[i].loc.x - puzzleObjects[0].loc.x;
        var diffY:int = puzzleObjects[i].loc.y - puzzleObjects[0].loc.y;
        // see if this object is appropriately placed
         // to be locked to the first one
        if (puzzleObjects[i].piece.x !=
                  (puzzleObjects[0].piece.x + pieceWidth*diffX)) return false;
         if (puzzleObjects[i].piece.y !=
                  (puzzleObjects[0].piece.y + pieceHeight*diffY)) return false;
     }
     return true;
}
```

The obligatory cleanup function benefits from our use of the two-sprite system. We can just remove these from the stage and set the variables referencing them to `null`. We also want to set the `puzzleObjects` and `beginDragged` to `null`, as well as the `ENTER_FRAME` and `MOUSE_UP` event:

```
public function cleanUpJigsaw() {
     removeChild(selectedPieces);
     removeChild(otherPieces);
     selectedPieces = null;
     otherPieces = null;
     puzzleObjects = null;
     beingDragged = null;
     removeEventListener(Event.ENTER_FRAME,movePieces);
     stage.removeEventListener(MouseEvent.MOUSE_UP,liftMouseUp);
}
```

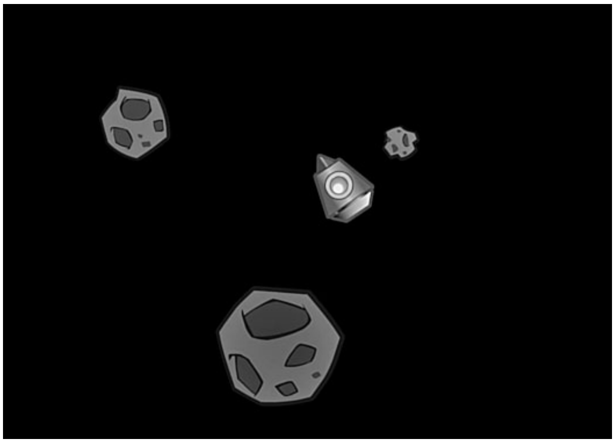
Modifying the Game

Game developers have come up with lots of ways to make computer jigsaw puzzles more interesting than their physical counterparts.

If you have played with creating bitmap filters using ActionScript, you might want to try doing that here. Applying glows, drop shadows, or bevels to the pieces can really make them pop.

You can also add piece rotation to make the puzzle harder. The pieces can be rotated 90, 180, or 270 degrees, and must be rotated back to 0 to fit in. Of course, you should also allow the player to rotate pieces after they are connected, which would add some challenging code to be able to rotate linked pieces together. Only attempt this if you are a ninja-level ActionScript programmer.

This page intentionally left blank



7

Direction and Movement: Space Rocks

- Using Math to Rotate and Move Objects
- Air Raid II
- Space Rocks

In Chapter 5, "Game Animation: Shooting and Bouncing Games," the games involved simple horizontal and vertical movement. Moving along the horizontal or vertical axis is very easy to program. But arcade games demand more.

In many games, you need to allow the player to turn and move. For instance, a driving game has both steering and forward movement. A space game also requires this, and might even need to allow the player to fire weapons in the direction that the player's ship is pointing.

Using Math to Rotate and Move Objects

SOURCE FILES
http://flashgameu.com
A3GPU07_RotationMath.zip

Combining rotation and movement means that we need to use deeper math than just addition, subtraction, multiplication, and division. We need to use basic trigonometry, such as sine, cosine, and arctangents.

If you're not into math, don't be scared. ActionScript does the hard part for us.

The Sin and Cos Functions

In Chapter 5, we used variables such as `dx` and `dy` to define the difference in horizontal and vertical positions. An object moving at a `dx` of 5 and a `dy` of 0 was moving 5 pixels to the right and 0 pixels up or down.

But how do we determine what `dx` and `dy` are if all we know is the rotation of an object? Suppose players have the ability to turn an object, like a car, in any direction. So, players point the car slightly down and to the right. Then, they go to move. You've got to change the `x` and `y` properties of the car, but you only know the angle at which the car is facing.

NOTE

The `rotation` property of any display object is a number between –180 and 180 representing the number of degrees that the object is turned from its original 0 degree rotation. You can change `rotation` just like you change the location values `x` and `y`. Rotation can also be more precise, like 23.76 degrees. So, if you want something to turn slowly, you can add .01 to it every frame or time period.

This is where the `Math.cos` and `Math.sin` functions come in. They allow us to compute `dx` and `dy` using only an angle.

Figure 7.1 shows the mathematics behind `Math.cos` and `Math.sin`. It is a graph of a circle. What `Math.cos` and `Math.sin` allow us to do is to find any point on the circle given the angle.

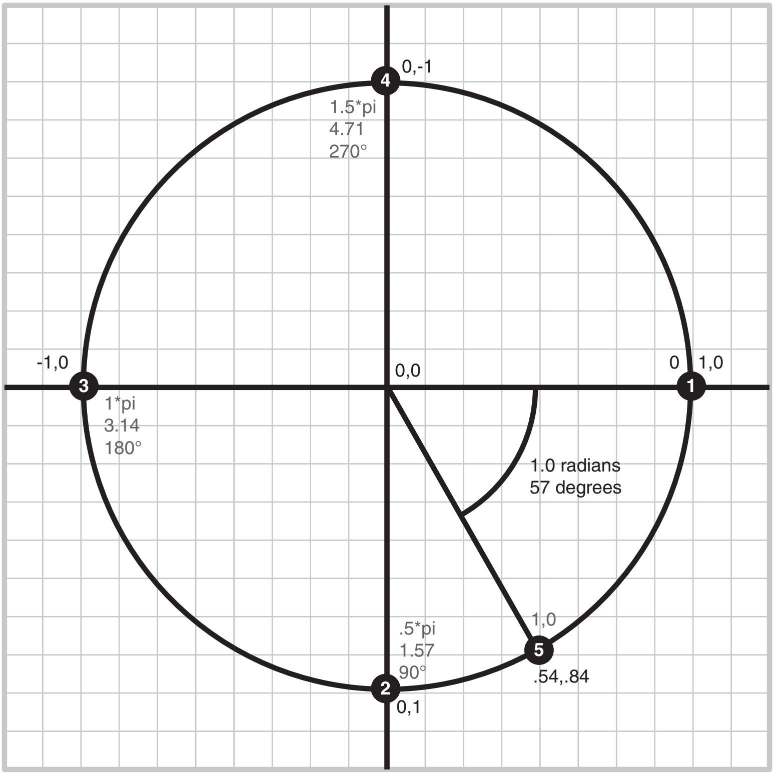


Figure 7.1
This graph of a circle shows the relationship between an angle and the x and y location of a point on the circle.

If the angle in question is 0, `Math.cos` and `Math.sin` return 1.0 and 0.0, respectively. This gives us point number 1, which has an `x` value of 1.0 and a `y` value of 0.0. So, an object rotated 0 degrees will move from the center of the circle to point 1.

If the object is pointed 90 degrees, `Math.cos` and `Math.sin` return 0.0 and 1.0, respectively. This is point number 2. An object pointed 90 degrees will move straight down.

Similarly, you can see where 180 degrees and 270 degrees lead: the first straight to the left, the second straight up.

NOTE

Figure 7.1 shows radians as a multiple of pi, the raw radians, and degrees. Radians and degrees are just two different ways of measuring angles. A complete circle is 360 degrees, which is 2 * pi radians. Pi is approximately 3.14, so 360 degrees = 6.26 radians.

ActionScript uses both degrees and radians. Degrees are used by the `rotation` property of an object. Radians are used by math functions such as `Math.cos` and `Math.sin`. So, we will constantly be converting back and forth from them.

These four directions are easy to figure out without the use of `Math.cos` and `Math.sin`. However, it is the angles in between them where we really rely on these trigonometry functions.

The 5th point is at an angle that is about 57 degrees. Determining where this is on the circle really does require `Math.cos` and `Math.sin`. The results are 0.54 in the `x` direction and 0.84 in the `y` direction. So, if an object were to move 1 pixel in distance while pointed 57 degrees, it would end up at point 5.

NOTE

It is important to realize that all 5 points, and in fact any point along the circle, are the exact same distance from the center. So, winding up at any of these points is not a matter of how fast the object is moving, but only a matter of what direction it is going.

Another important thing to remember is that `Math.cos` and `Math.sin` always return values between –1.0 and 1.0. It assumes that the circle is 1.0 units in radius. So, if an object is at 57 degrees and moves 1.0 units, it will move to 0.54,0.84. However, if it has a speed of 5, we multiply that by 5 and get 2.70,4.20 as the amount moved.

Using Cos and Sin to Drive a Car

A simple example helps to explain the use of these trigonometry functions. The movies **`MovingCar.fla`** and **`MovingCar.as`** act as a basic driving simulation. A car is placed in the middle of the screen, and the player can use the left- and right-arrow keys to turn, and the up arrow to move forward. Figure 7.2 shows the car on the screen.

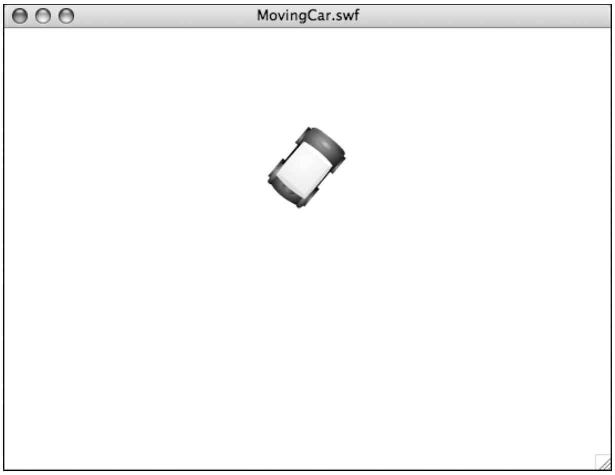


Figure 7.2
A simple driving demonstration allows the player to steer and move.

We'll use some code similar to the Air Raid game of Chapter 5. There will be three Boolean variables, `leftArrow`, `rightArrow`, and `upArrow`. All of these will be set to `true` when players press the associated key, and `false` when they lift the key back up.

Here is the start of the class, with the listeners and the code to handle the arrow keys. Notice that we don't need any extra imports to use the `Math` functions. These are part of the standard ActionScript library:

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

    public class MovingCar extends MovieClip {
        private var leftArrow, rightArrow, upArrow: Boolean;

        public function MovingCar() {

            // move every frame
            addEventListener(Event.ENTER_FRAME, moveCar);

            // respond to key events
            stage.addEventListener(KeyboardEvent.KEY_DOWN,keyPressedDown);
            stage.addEventListener(KeyboardEvent.KEY_UP,keyPressedUp);
        }

        // set arrow variables to true
        public function keyPressedDown(event:KeyboardEvent) {
            if (event.keyCode == 37) {
                leftArrow = true;
            } else if (event.keyCode == 39) {
                rightArrow = true;
            } else if (event.keyCode == 38) {
                upArrow = true;
            }
        }

        // set arrow variables to false
        public function keyPressedUp(event:KeyboardEvent) {
            if (event.keyCode == 37) {
                leftArrow = false;
            } else if (event.keyCode == 39) {
                rightArrow = false;
            } else if (event.keyCode == 38) {
                upArrow = false;
            }
        }
```

On every frame, the `moveCar` function is called. It looks at each of the Boolean values and determines what to do if any are `true`. In the case of the left and right arrows, the `rotation` property of the car movie clip is changed, so the car rotates.

NOTE

Note that we are not using time-based animation here. So, setting the frame rate of your movie to different values will change the speed of rotation and travel.

If the up arrow is pressed, the `moveForward` function is called:

```
// turn or move car forward
public function moveCar(event:Event) {
     if (leftArrow) {
          car.rotation -= 5;
     }
     if (rightArrow) {
          car.rotation += 5;
     }
     if (upArrow) {
          moveForward();
     }
}
```

This is where we get to use our math. If the up arrow is pressed, we first calculate the angle, in radians, of the car. We know the rotation of the car, but that is in degrees. To convert degrees to radians, we divide by 360 (the number of degrees in a circle), and then multiply by twice pi (the number of radians in a circle). We'll be using this conversion often, so it is worth breaking it down for clarity:

1. Divide by 360 to convert the 0 to 360 value to a 0 to 1.0 value.
2. Multiply by 2 * pi to convert the 0 to 1.0 value to a 0 to 6.28 value.

```
radians = 2 * pi * (degrees / 360)
```

Conversely, when we want to convert radians to degrees, we do this:

1. Divide by 2 * pi to convert the 0 to 6.28 value to a 0 to 1.0 value.
2. Multiply by 360 to convert the 0 to 1.0 value to a 0 to 360 value.

```
degrees = 360 * radians / (2 * pi)
```

NOTE

Because both degrees and radians measure angles, the values repeat themselves every 360 degrees or 2 * pi radians. So, 0 degrees and 360 degrees are the same; 90 and 450 degrees are also the same. This even works with negative values. For example, 270 degrees and –90 degrees are the same. In fact, the `rotation` property of any display object always returns a value from –180 to 180, which is the same as pi and -pi radians.

Now that we have the angle in radians, we feed it into `Math.cos` and `Math.sin` to get the `dx` and `dy` values for movement. We also multiply by `speed`, a value we set earlier in the function. This moves the car 5 pixels per frame, rather than 1 pixel per frame.

Finally, we change the `x` and `y` properties of the car to actually move it:

```
        // calculate x and y speed and move car
        public function moveForward() {
            var speed:Number = 5.0;
            var angle:Number = 2*Math.PI*(car.rotation/360);
            var dx:Number = speed*Math.cos(angle);
            var dy:Number = speed*Math.sin(angle);
            car.x += dx;
            car.y += dy;
        }
    }
}
```

Play with the **`MovingCar.fla`** movie. Turn the car to different angles and press the up arrow to see it move. Picture the `Math.cos` and `Math.sin` functions translating the angle to an amount of horizontal and vertical movement.

Then, have some fun. Press down the left- and up-arrow keys at the same time to make the car go in circles. This is the same effect as turning your steering wheel on your real car and pressing the gas. The car continues to turn.

Forgetting for a minute about acceleration, we've got a pretty fun little car simulation going. In Chapter 12, "Game Worlds: Driving and Exploration Game," we actually build a much more complex driving simulation, but the basic use of `Math.cos` and `Math.sin` are at the heart of it.

Calculating an Angle from a Location

Although `Math.sin` and `Math.cos` allow you to get x and y coordinates from an angle, we also occasionally need to get an angle from a set of x and y coordinates. To do this, we use an arctangent calculation. The ActionScript function for this is `Math.atan2`.

Figure 7.3 shows how the arctangent function works. Point 1 is located at 6,5 on the grid. To find its angle, we take the y distance and the x distance and feed them in to `Math.atan2`. The result would be .69 radians, or about 40 degrees.

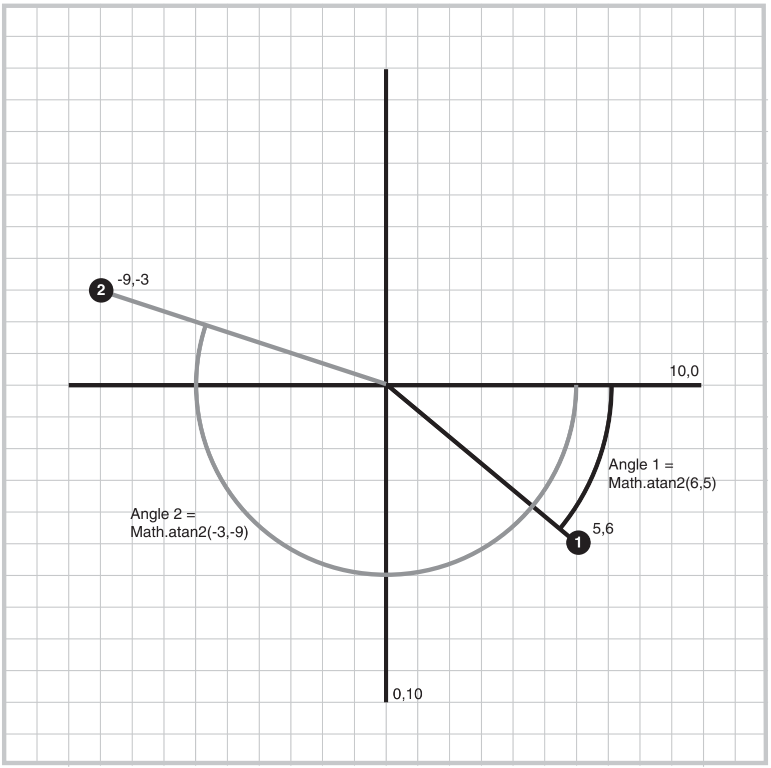


Figure 7.3
The angles of these two points can be determined by using `Math.atan2`.

The second point is at –9,–3. Feeding that into `Math.atan2` gives us –2.82 radians, or –162 degrees. That is the same as 198 degrees. The `Math.atan2` function likes to keep numbers between –180 and 180.

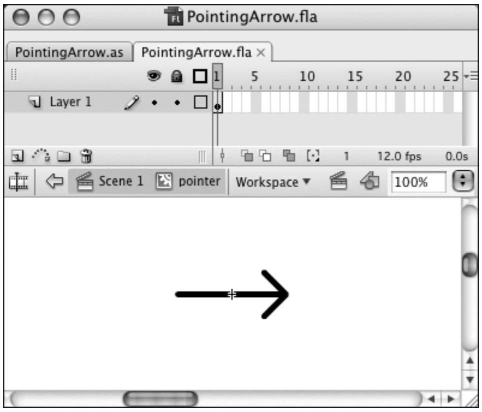
NOTE

There is also a `Math.atan` function. This takes one parameter: the ratio of the y distance over the x distance. So, you would use it like `Math.atan(dy/dx)`. This is the traditional arctangent mathematical function. The problem with it is that you don't know whether the result is forward or backward. For instance, –5/3 is the same as 5/–3. One is 121 degrees, whereas the other is –60 degrees. The `Math.atan` function returns –60 degrees for both. The `Math.atan2` function gives you the correct angle.

We can create a simple example using an arrow. You can find it in the source files **PointingArrow.fla** and **PointingArrow.as**.

The arrow is located at the center of the screen (location 275,200). Look at Figure 7.4 and notice that the registration point for the movie clip is at the center of the arrow. When you rotate a movie clip, it rotates around this point. Also, notice that the arrow is pointing due right. A rotation of 0 corresponds to that direction, so any object created with the sole purpose of being rotated should be created facing right like this.

Figure 7.4
It is easier to rotate objects that start off facing right, with the center of the movie clip at the center of rotation.



This pointer will point "to" the cursor. So, we have an origin point for the pointer of 275,200 and a destination point of the cursor location. Because it is easy to move the cursor and change `mouseX` and `mouseY`, this is a quick way to experiment with `Math.atan2`.

The following short class, from **`PointingArrow.as`**, calls a function every frame. This function computes the `dx` and `dy` values from the distance between the cursor and the pointer's location. It then uses `Math.atan2` to compute the angle in radians. It converts that to degrees and sets the `rotation` property of the pointer with it:

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

    public class PointingArrow extends MovieClip {

        public function PointingArrow() {
            addEventListener(Event.ENTER_FRAME, pointAtCursor);
        }

        public function pointAtCursor(event:Event) {
            // get relative mouse location
            var dx:Number = mouseX - pointer.x;
            var dy:Number = mouseY - pointer.y;

            // determine angle, convert to degrees
            var cursorAngle:Number = Math.atan2(dy,dx);
            var cursorDegrees:Number = 360*(cursorAngle/(2*Math.PI));

            // point at cursor
            pointer.rotation = cursorDegrees;
        }
    }
}
```

When you run this movie, the pointer points at the cursor at all times, as you can see in Figure 7.5. Or, at least while `mouseX` and `mouseY` are updating, which is only when the cursor is over the Flash movie.

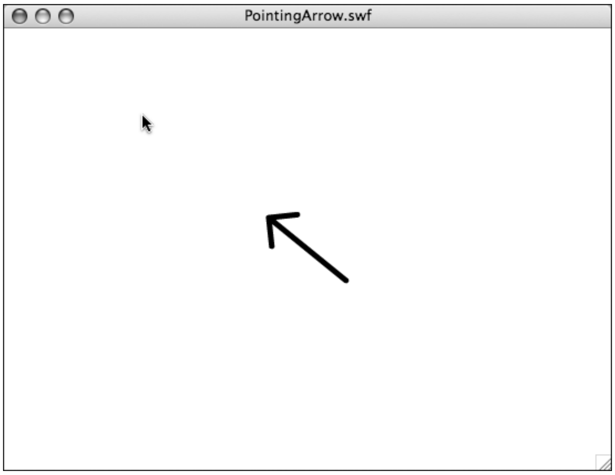


Figure 7.5
The arrow points at the cursor as long as the cursor is over the movie.

NOTE
From these two simple examples can come some interesting results if you combine them. For instance, what if the car were steered by the location of the mouse relative to the car? The car would point at the mouse, and then when you move, it would move toward the mouse at all times. It would, essentially, chase the mouse. So, what if the player were to drive the car like in the first example, but a second car points at the mouse and drives by itself? The second car would chase the first car! See http://flashgameu.com for an example.

Now that you know how to use trigonometry to observe and control the positions and movement of objects, we can apply these to some games.

Air Raid II

SOURCE FILES
http://flashgameu.com
A3GPU07_AirRaid2.zip

In Chapter 5's Air Raid game, you moved an anti-aircraft gun back and forth using the arrow keys. This allowed you to aim at different parts of the sky as you shot upward.

Now with the power of `Math.sin` and `Math.cos`, we can change this game to keep the gun stationary, but allow it to aim at an angle to hit different targets.

Altering the Gun

The first thing we need to do is to change the `AAGun` movie clip to allow for rotating gun barrels. We'll take the base of the turret out of the movie clip completely, and place it in its own movie clip, `AAGunBase`. The gun barrels will remain in `AAGun`, but we'll recenter it so that the pivot point is at the center and the barrels point to the right, as in Figure 7.6.

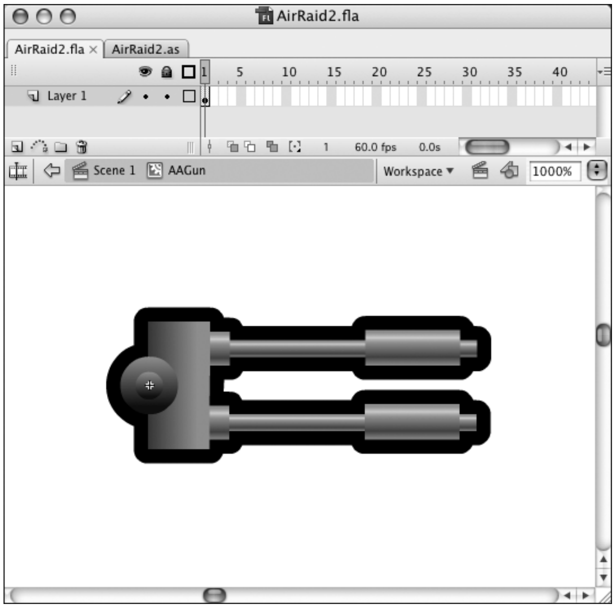


Figure 7.6
The barrels must point to the right to correspond with cos and sin values.

The idea is to change the original Air Raid game as little as possible. We'll be taking the same values for arrow-key presses and using them to change the `rotation` property of the AAGun, rather than the `y` value.

NOTE
Alternatively, you could have a different set of keys set the rotation (for example, A and S or the command and period). Then, leave the arrow keys to move the gun, and you could have both a moving gun and a rotating barrel.

The `x` and `y` values of the gun are still set, but the `rotation` value is also set, to –90. The value of –90 means that the gun starts pointed straight up. We'll restrict the value of the rotation in the same way that we restricted horizontal movement in the first version of Air Raid. In this case, the values stay between –170 and –20 degrees, which is 50 degrees to the left or right of straight up.

So, here is our new **AAGun.as** code. Look for the lines in the following code that involve the `newRotation` variable and `rotation` property:

```
package {
    import flash.display.*;
    import flash.events.*;
    import flash.utils.getTimer;

    public class AAGun extends MovieClip {
        static const speed:Number = 150.0;
        private var lastTime:int; // animation time

        public function AAGun() {
            // initial location of gun
            this.x = 275;
            this.y = 340;
            this.rotation = -90;

            // movement
            addEventListener(Event.ENTER_FRAME,moveGun);
        }

        public function moveGun(event:Event) {
            // get time difference
            var timePassed:int = getTimer()-lastTime;
            lastTime += timePassed;

            // current position
            var newRotation = this.rotation;

            // move to the left
            if (MovieClip(parent).leftArrow) {
                newRotation -= speed*timePassed/1000;
            }

            // move to the right
            if (MovieClip(parent).rightArrow) {
                newRotation += speed*timePassed/1000;
            }
            // check boundaries
            if (newRotation < -170) newRotation = -170;
            if (newRotation > -20) newRotation = -20;

            // reposition
            this.rotation = newRotation;
        }
        // remove from screen and remove events
        public function deleteGun() {
            parent.removeChild(this);
            removeEventListener(Event.ENTER_FRAME,moveGun);
        }
    }
}
```

Notice that the `speed` value of 150 stays the same. It is very likely that switching from horizontal movement to rotational movement would mean a change in the `speed` value, but in this case the value of 150 works well for both.

Changing the Bullets

The **`Bullets.as`** class needs to change to have the bullets move upward at an angle, rather than straight up.

The graphic must change, too. The bullets need to point to the right, and they should be centered on the registration point. Figure 7.7 shows the new `Bullet` movie clip.

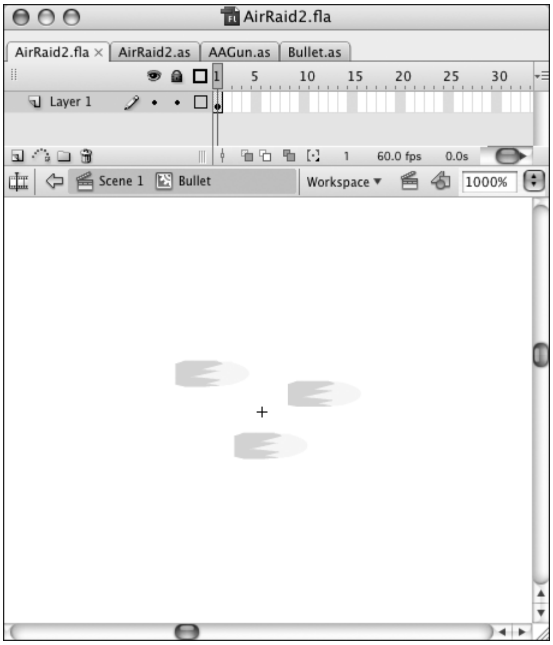


Figure 7.7
The new `Bullet` *movie clip re-centers the graphic and points it to the right.*

The class needs to change to add both `dx` and `dy` movement variables. They will be calculated from the angle at which the bullet was fired, which is a new parameter passed into the `Bullet` function.

In addition, the bullet needs to start off at some distance from the center of the gun; in this case, it should be 40 pixels away from center. So, the `Math.cos` and `Math.sin` values are used both to compute the original position of the bullet and to compute the `dx` and `dy` values.

Also, the rotation of the `Bullet` movie clip will be set to match the rotation of the gun. So, the bullets will start just above the end of the turret, pointed away from the turret, and continue to move directly away at the same angle:

```
package {
     import flash.display.*;
     import flash.events.*;
     import flash.utils.getTimer;

     public class Bullet extends MovieClip {
          private var dx,dy:Number; // speed
          private var lastTime:int;

          public function Bullet(x,y:Number, rot: Number, speed: Number) {
               // set start position
               var initialMove:Number = 35.0;
               this.x = x + initialMove*Math.cos(2*Math.PI*rot/360);
               this.y = y + initialMove*Math.sin(2*Math.PI*rot/360);
               this.rotation = rot;

               // get speed
               dx = speed*Math.cos(2*Math.PI*rot/360);
               dy = speed*Math.sin(2*Math.PI*rot/360);

               // set up animation
               lastTime = getTimer();
               addEventListener(Event.ENTER_FRAME,moveBullet);
          }

          public function moveBullet(event:Event) {
               // get time passed
               var timePassed:int = getTimer()-lastTime;
               lastTime += timePassed;

               // move bullet
               this.x += dx*timePassed/1000;
               this.y += dy*timePassed/1000;

               // bullet past top of screen
               if (this.y < 0) {
                    deleteBullet();
               }
```

```
        }

        // delete bullet from stage and plane list
        public function deleteBullet() {
            MovieClip(parent).removeBullet(this);
            parent.removeChild(this);
            removeEventListener(Event.ENTER_FRAME,moveBullet);
        }

    }
}
```

Changes to AirRaid2.as

Changes are needed to the main class to facilitate the new versions of `AAGun` and `Bullet`. Let's look at each change. We'll be creating a new class called **`AirRaid2.as`** and changing the movie's document class to match it. Remember to also change the class definition at the top of the code to be `AirRaid2` rather than `AirRaid`.

In the class variable definitions, we need to add the new `AAGunBase` movie clip as well as keep the `AAGun` movie clip:

```
private var aagun:AAGun;
private var aagunbase:AAGunBase;
```

In `startAirRaid`, we need to account for the fact that there are two movie clips representing the gun, too. The `AAGunBase` does not have a class of its own, so we need to set its position to match that of the `AAGun`.

NOTE

You could also remove the `AAGunBase` entirely by using a different design, or seating the barrels into a graphic that exists at part of the background.

```
// create gun
aagun = new AAGun();
addChild(aagun);
aagunbase = new AAGunBase();
addChild(aagunbase);
aagunbase.x = aagun.x;
aagunbase.y = aagun.y;
```

The only other necessary change is down in the `fireBullet` function. This function needs to pass on the rotation of the gun to the `Bullet` class, so that it knows what direction to shoot the bullet at. So, we'll add that third parameter to match the third parameter in the `Bullet` function that creates a new bullet:

```
var b:Bullet = new Bullet(aagun.x,aagun.y,aagun.rotation,300);
```

NOTE

If we were building this game from scratch, we might not even include the first two parameters, which refer to the position of the gun. After all, the gun won't be moving, so it will always remain at the same position. Because we already had code that dealt with relating the bullet start point to the gun position, we can leave it in and gain the benefit of having only one place in the code where the gun position is set.

We've succeeded in changing the **`AirRaid2.as`** class. In fact, if we hadn't needed to add the cosmetic `AAGunBase` to the movie, we would have only needed that last change in **`AirRaid2.as`**. This demonstrates how versatile ActionScript can be if you set it up with a different class for each moving element.

Now we have a fully transformed Air Raid II game that uses a rotating, but stationary gun.

Space Rocks

SOURCE FILES
http://flashgameu.com
A3GPU07_SpaceRocks.zip

One of the most classic video games of all time was Asteroids. This vector-based arcade game was released by Atari in 1979. It featured simple single-colored lines for graphics, very basic sound, and easy ways to cheat and win. Despite this, the game was very addictive due to great basic game play.

In the game, you controlled a small spaceship. You could turn, shoot, and fly around the screen. Against you were a few large asteroids moving at random speed and directions. You could break them apart into smaller asteroids by shooting at them. The smallest asteroids would disappear when shot. If an asteroid hit you, you lost a life.

We'll build a game with the same basic concepts: a spaceship, rocks, and missiles. We'll even use one of the more advanced features of the original game: a shield.

Game Elements and Design

Before we start, we need to decide what our game, Space Rocks, will be like. We don't need to create a complete design document, but a few lists will help us stay focused as we build the game from scratch.

The game elements are a ship, rocks, and missiles. You can see them, in all variations, in Figure 7.8.

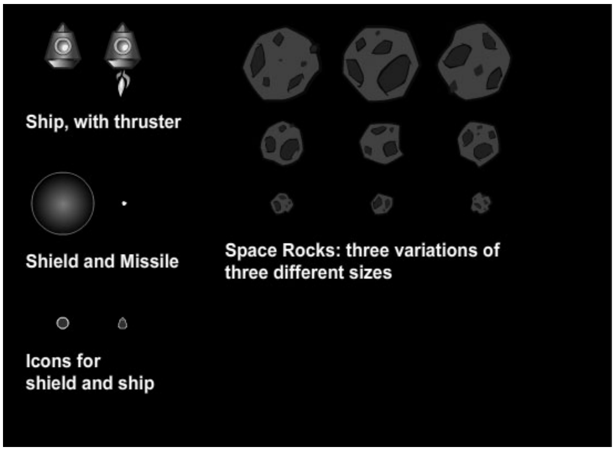

Ship, with thruster
Shield and Missile
Icons for shield and ship
Space Rocks: three variations of three different sizes

Figure 7.8
Here are all the game elements for Space Rocks.

Let's look at the abilities of the ship. Here is a list of what the ship can do:

- Appears stationary in the middle of the screen to start
- Turns left when the left arrow is pressed
- Turns right when the right arrow is pressed
- Accelerates forward when the up arrow is pressed
- Moves according to its velocity
- Generates a shield when the Z key is pressed

The ship fires a missile. Here's what the missiles will do:

- Created when the player presses the spacebar
- Velocity and position determined by location and rotation of ship
- Move according to its velocity

Rocks do the following:

- Have a random starting velocity and rotation speed
- Move according to its velocity
- Rotate according to a rotation speed
- Have three different sizes: big, medium and small

Collisions are what this game is all about. There are two types of collisions: missile with rock, and rock with ship.

When a missile and a rock collide, the original rock is removed. If it was a “big” rock, two medium rocks appear at the same location. If it was a “medium” rock, two “small” rocks appear at the same location. Small rocks just disappear, no rocks replace them. The missile in a collision will also be removed.

When a rock and a ship collide, the rock behaves like a missile hit it. The ship is removed. The player has three lives. If this isn’t the last life for the player, he gets another ship, which appears in the center of the screen, after two seconds pass.

If the player shoots all the rocks, and there are none left on the screen, the level is over. After a short delay, a new wave of rocks appears, but a little faster than the last set.

NOTE

In most of the 1970s versions of Asteroids, there was a maximum speed cap on the speed of the rocks. This allowed an expert player to continue to play indefinitely, or until the arcade closed or the player’s mom insisted it was time for dinner.

Another action the player can take is to generate a shield. Pressing the Z key creates a shield around the ship for three seconds. This makes the ship able to pass through rocks. But, players only have three shields per life. So, they must use them carefully.

One important aspect of the game is that both the ship and rocks wrap around the screen while moving. If one of them goes off the screen to the left, it appears again on the right. If one goes off the bottom, it appears again on the top. The missiles, however, just travel to the edge of the screen and disappear.

Setting Up the Graphics

We need a ship, some rocks, and a missile to create this game. The ship is the most complex element. It needs a plain state, a state with a thruster turned on, and some sort of explosion animation for it when it is hit. It also needs a shield that covers the ship at times.

Figure 7.9 shows a movie clip of the ship exploding. There are several frames. The first is the ship without thrusters, and the second is the ship with thrusters. The rest of the frames are a short explosion animation.

The shields are actually another movie clip placed inside the ship movie clip. It is present on both the first (no thrusters) and second (thrusters) frames. We’ll turn shields off by setting its `visible` property to `false`. And then when we need them, we’ll turn the `visible` property to `true`.

The rocks will actually be a series of movie clips. There will be three for the three sizes: `Rock_Big`, `Rock_Medium`, and `Rock_Small`. All three movie clips will in turn have three frames representing three variations of the rocks. This prevents all the rocks from looking the same. Figure 7.10 shows the `Rock_Big` movie clip, and you can see the keyframes containing the three variations up in the timeline.

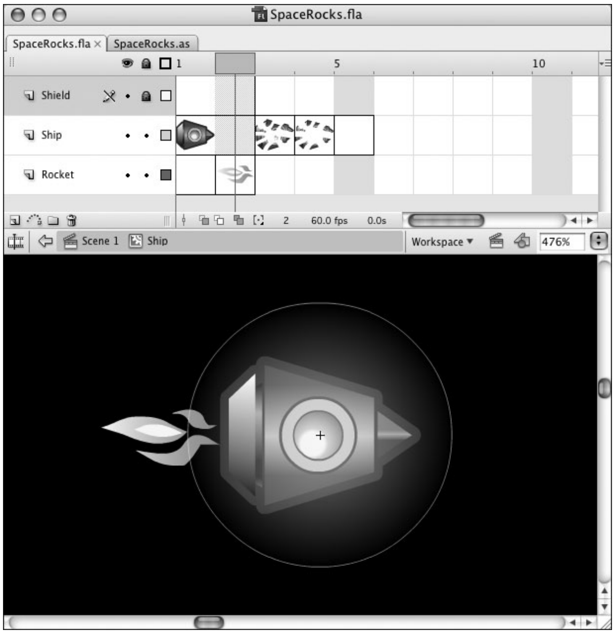


Figure 7.9
This frame of the ship has both thrusters and shields turned on.

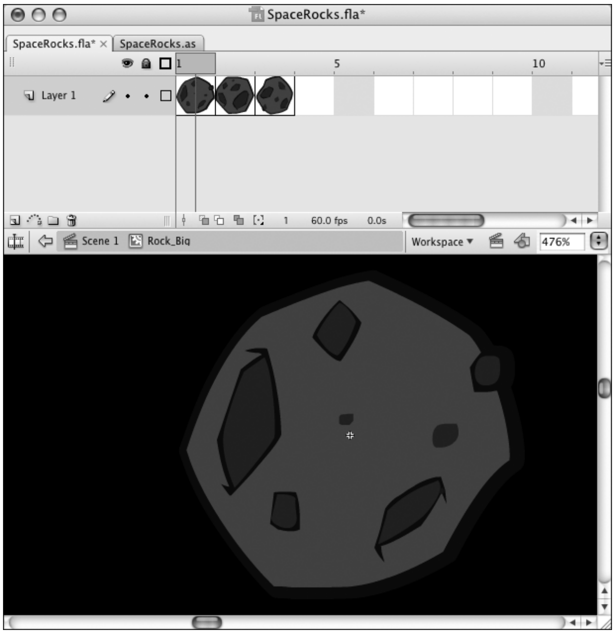


Figure 7.10
Each movie clip for the rocks has three variations of rocks, all the same size.

The missile is the simplest element. It is only a small yellow dot. There are also two other movie clips: `ShipIcon` and `ShieldIcon`. These are small versions of the ship, and a shield. We'll use these to display the number of ships and shields remaining.

The main timeline is set up in the typical way: three frames with the middle frame calling `startSpaceRocks`. Now we just need to create the ActionScript to make the game come alive.

Setting Up the Class

We'll place all the code in one **`SpaceRocks.as`** class file. This will make for the longest class file in this book so far. The advantage of a single class file here is that all of our code is in one place. The disadvantage is that it can get long and unwieldy.

To help, we'll break up the code into smaller sections, each one dealing with a different screen element. But first, let's look at the class declaration.

The class needs a typical set of imports to handle all the different objects and structures:

```
package {
     import flash.display.*;
     import flash.events.*;
     import flash.text.*;
     import flash.utils.getTimer;
     import flash.utils.Timer;
     import flash.geom.Point;
```

A host of constants allow you to tweak the feeling and difficulty of the game. The speeds are all measured in units per thousandths of a second, so the `shipRotationSpeed` is a pretty fast .1/1000 or 100 degrees per second. The missiles will move at 200 pixels per second, and the thrusters will accelerate the ship at 150 pixels per second per second.

NOTE

Speed is measured in units per time, such as 100 pixels per second. Acceleration is the change in speed over time: how many pixels per second the speed changes per second. So, we can say: pixels per second per second.

The speed of the rocks will depend on the level. It will be .03 plus .02 times the level—so, .05 for the first level, .07 for the second, and so on.

We also lock-in the radius of the ship, which is kind of round in shape. We'll use this radius to detect a collision, instead of relying on the `hitTestObject` function:

```
     public class SpaceRocks extends MovieClip {
          static const shipRotationSpeed:Number = .1;
          static const rockSpeedStart:Number = .03;
          static const rockSpeedIncrease:Number = .02;
          static const missileSpeed:Number = .2;
          static const thrustPower:Number = .15;
          static const shipRadius:Number = 20;
          static const startingShips:uint = 3;
```

After the constants, we need to define a bunch of variables to be set later. Here are the variables that hold references to the ship, rocks, and missiles:

```
// game objects
private var ship:Ship;
private var rocks:Array;
private var missiles:Array;
```

Then, we have an animation timer that will be used to keep all movement in step:

```
// animation timer
private var lastTime:uint;
```

The left-, right-, and up-arrow keys will be tracked by the following Boolean values:

```
// arrow keys
private var rightArrow:Boolean = false;
private var leftArrow:Boolean = false;
private var upArrow:Boolean = false;
```

Ship velocity will be broken into two speed values:

```
// ship velocity
private var shipMoveX:Number;
private var shipMoveY:Number;
```

We have two timers. One is the delay after the player loses a ship, before the next one appears. We'll also use it to delay the next set of rocks after all the rocks have been destroyed. The other is the length of time a shield will last:

```
// timers
private var delayTimer:Timer;
private var shieldTimer:Timer;
```

There is a `gameMode` variable that can be set to either `"play"` or `"delay"`. When it is `"delay"`, we won't listen to key presses from the player. We also have a Boolean that tells us whether the shield is on, and the player can't be hurt by rocks:

```
// game mode
private var gameMode:String;
private var shieldOn:Boolean;
```

The next set of variables deal with the shields and ships. The first two are numbers that track the number of ships and shields; the second two are arrays that hold references to the icons displayed on the screen that relay this information to the player:

```
// ships and shields
private var shipsLeft:uint;
private var shieldsLeft:uint;
private var shipIcons:Array;
private var shieldIcons:Array;
```

The score is stored in `gameScore`. It is displayed to the player in a text field we'll create named `scoreDisplay`. The `gameLevel` variable keeps track of the number of sets of rocks that have been cleared:

```
        // score and level
        private var gameScore:Number;
        private var scoreDisplay:TextField;
        private var gameLevel:uint;
```

Finally, we have two sprites. We'll be placing all the game elements in these two sprites. The first is `gameObjects`, and will be our main sprite. But, we'll place the ship and shield icons, and the score in the `scoreObjects` sprite to separate them:

```
        // sprites
        private var gameObjects:Sprite;
        private var scoreObjects:Sprite;
```

Starting the Game

The constructor function will start by setting up the sprites. It is important that the `addChild` statements appear in this order so that the icons and score stay above the game elements:

```
 // start the game
 public function startSpaceRocks() {
      // set up sprites
      gameObjects = new Sprite();
      addChild(gameObjects);
      scoreObjects = new Sprite();
      addChild(scoreObjects);
```

The `gameLevel` is set to 1, and the `shipsLeft` is set to 3, which comes from the constants defined earlier. The `gameScore` is zeroed out, too. Then, a call to `createShipIcons` and `createScoreDisplay` will set those things up. We'll see them soon:

```
      // reset score objects
      gameLevel = 1;
      shipsLeft = startingShips;
      gameScore = 0;
      createShipIcons();
      createScoreDisplay();
```

We need three listeners, similar to the Air Raid games. One will be a general frame function call; the other two deal with key presses:

```
      // set up listeners
      addEventListener(Event.ENTER_FRAME,moveGameObjects);
      stage.addEventListener(KeyboardEvent.KEY_DOWN,keyDownFunction);
      stage.addEventListener(KeyboardEvent.KEY_UP,keyUpFunction);
```

To kick the game off, we set the `gameMode` to `"delay"` and the `shieldOn` to `false`, create an array for the missiles to be stored in, and then call two functions to start the game. The first creates the first set of rocks, and the second creates the first ship. Because both of these functions will later be called by event timers, we need to include `null` as a parameter here to fill the spot that the event timer value will use later:

```
    // start
    gameMode = "delay";
    shieldOn = false;
    missiles = new Array();
    nextRockWave(null);
    newShip(null);
}
```

Score and Status Display Objects

The first large group of functions deals with the number of ships the player has, the number of shields the player has, and the player's score. These display in three corners of the screen.

The score is shown as text in the upper right. The number of ships left is shown by zero to three ship icons in the lower left. The number of shields left is shown by zero to three shield icons in the lower right. Figure 7.11 shows the game at the start with all three items present.

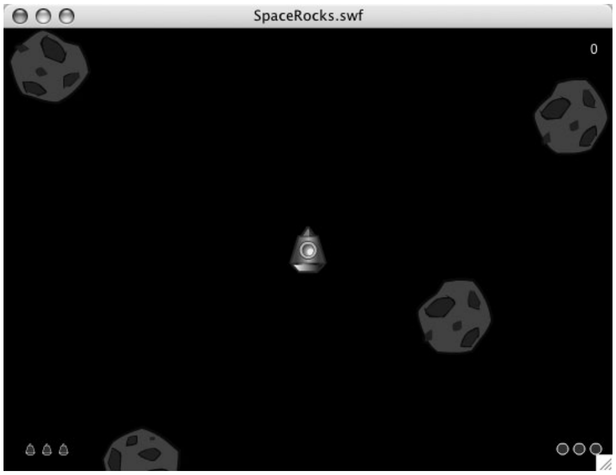


Figure 7.11 *The score is in the upper right, the number of lives in the bottom left, and the number of shields remaining in the lower right.*

To create the ship and shield icons, the next two functions loop and place the three items on the screen. They are added to their respective arrays so that they can be references and removed later:

```
// draw number of ships left
public function createShipIcons() {
    shipIcons = new Array();
    for(var i:uint=0;i<shipsLeft;i++) {
        var newShip:ShipIcon = new ShipIcon();
        newShip.x = 20+i*15;
        newShip.y = 375;
        scoreObjects.addChild(newShip);
        shipIcons. push(newShip);
    }
}
```

Here is a similar function for the shield icons:

```
// draw number of shields left
public function createShieldIcons() {
    shieldIcons = new Array();
    for(var i:uint=0;i<shieldsLeft;i++) {
        var newShield:ShieldIcon = new ShieldIcon();
        newShield.x = 530-i*15;
        newShield.y = 375;
        scoreObjects.addChild(newShield);
        shieldIcons.push(newShield);
    }
}
```

NOTE

We could also have avoided the icons altogether and just used text fields to display the number of ships and shields remaining. This would be less code, but not as visually interesting.

Creating the score display is a matter of making a new text field and setting its properties. We also create a temporary `TextFormat` variable and use that to set the `defaultTextFormat` of the field:

```
// put the numeric score at the upper right
public function createScoreDisplay() {
    scoreDisplay = new TextField();
    scoreDisplay.x = 500;
    scoreDisplay.y = 10;
    scoreDisplay.width = 40;
    scoreDisplay.selectable = false;
    var scoreDisplayFormat = new TextFormat();
    scoreDisplayFormat.color = 0xFFFFFF;
    scoreDisplayFormat.font = "Arial";
    scoreDisplayFormat.align = "right";
    scoreDisplay.defaultTextFormat = scoreDisplayFormat;
    scoreObjects.addChild(scoreDisplay);
```

```
     updateScore();
}
```

At the end of `createScoreDisplay`, we call `updateScore` immediately to put a `0` into the field, because that is the value of `gameScore` at this point. But, the `updateScore` function will be used later, too, any time we have a change in the score:

```
// new score to show
public function updateScore() {
     scoreDisplay.text = String(gameScore);
}
```

When it comes time to remove a ship or a shield, we need to `pop` an item from the `shipIcons` or `shieldIcons` arrays and `removeChild` from the `scoreObjects` to erase the icon:

```
// remove a ship icon
public function removeShipIcon() {
     scoreObjects.removeChild(shipIcons.pop());
}

// remove a shield icon
public function removeShieldIcon() {
     scoreObjects.removeChild(shieldIcons.pop());
}
```

We should also add functions that loop and remove all the icons. We need this at the end of the game; and for the shields, we need it at the end of a life. We want to give the player a full three shields with every new ship, so we'll just delete the shield icons and start over again when that happens:

```
// remove the rest of the ship icons
public function removeAllShipIcons() {
     while (shipIcons.length > 0) {
          removeShipIcon();
     }
}

// remove the rest of the shield icons
public function removeAllShieldIcons() {
     while (shieldIcons.length > 0) {
          removeShieldIcon();
     }
}
```

Ship Movement and Player Input

The next set of functions all deal with the ship. The first function creates a new ship. The rest of the functions deal with moving the ship.

Creating a New Ship

The `newShip` function is called at the start of the game, and it is also called two seconds after the previous ship's demise. On those subsequent times, it will be a timer that does the calling, so a `TimerEvent` is passed to it. We won't need it for anything, however.

On the 2nd, 3rd, and 4th times, the function is called, the previous ship still exists. It will have played out its explosion animation. At the end of this animation, a simple `stop` command pauses the movie clip at the last frame, which is blank. So, the ship is still there, just invisible. We'll look for the ship to be something other than `null`, and then remove the ship and clear it out before doing anything else.

NOTE

In other games, it might be desirable to remove the ship as soon as the explosion animation is over. In that case, you can just place a call back to the main class from within the ship timeline. This call can be on the last frame of the animation, so you know the animation is over and the object can be removed.

```
// create a new ship
public function newShip(event:TimerEvent) {
     // if ship exists, remove it
     if (ship != null) {
          gameObjects.removeChild(ship);
          ship = null;
     }
```

Next, we check to see whether any ships are left. If not, the game is over:

```
     // no more ships
     if (shipsLeft < 1) {
          endGame();
          return;
     }
```

A new ship is created, positioned, and set to the first frame, which is the plain ship with no thruster. The rotation is set to –90, which will point it straight up. We also need to remove the shield. Then, we can add the movie clip to the `gameObjects` sprite:

```
     // create, position, and add new ship
     ship = new Ship();
     ship.gotoAndStop(1);
     ship.x = 275;
     ship.y = 200;
```

```
    ship.rotation = -90;
    ship.shield.visible = false;
    gameObjects.addChild(ship);
```

The velocity of the ship is stored in the `shipMoveX` and `shipMoveY` variables. And now that we've created a ship, the `gameMode` can be changed from `"delay"` to `"play"`:

```
    // set up ship properties
    shipMoveX = 0.0;
    shipMoveY = 0.0;
    gameMode = "play";
```

With every new ship, we reset the shields to 3. Then, we need to draw the three little shield icons at the bottom of the screen:

```
    // set up shields
    shieldsLeft = 3;
    createShieldIcons();
```

When the player loses a ship, and a new ship appears, there is a chance that it will reappear in the middle of the screen at exactly the moment that a rock passes by. To prevent this, we can use the shields. By turning the shields on, the player is guaranteed to be collision-free for three seconds.

NOTE

The original arcade games of this type avoided the problem of having a ship appear in the middle of a rock by simply waiting until the middle of the screen was relatively empty before creating a new ship. You could do this, too, by checking the distance of each rock to the new ship, and just delaying another two seconds if anything is close.

Note that we only want to do this if this is not the first time the ship appears. The first time it appears, the rocks will also be making their first appearance, which are at preset locations away from the center.

When we call `startShield` here, we pass the value `true` to it to indicate that this is a free shield. It won't be charged against the player's allotted three shields per ship:

```
    // all lives but the first start with a free shield
    if (shipsLeft != startingShips) {
        startShield(true);
    }
}
```

Handing Keyboard Input

The next two functions take care of key presses. As with Air Raid, we track the left and right arrows. We also care about the up arrow. In addition, we react to the spacebar and the Z key, when they are pressed.

In the case of the up arrow, we also turn on the thruster by telling the ship to go to the second frame, where the thruster image is located.

A spacebar calls `newMissile`, and a Z calls `startShield`:

```
// register key presses
public function keyDownFunction(event:KeyboardEvent) {
    if (event.keyCode == 37) {
        leftArrow = true;
    } else if (event.keyCode == 39) {
        rightArrow = true;
    } else if (event.keyCode == 38) {
        upArrow = true;
        // show thruster
        if (gameMode == "play") ship.gotoAndStop(2);
    } else if (event.keyCode == 32) { // space
        newMissile();
    } else if (event.keyCode == 90) { // z
        startShield(false);
    }
}
```

The `keyUpFunction` turns the thruster off when the player lifts up on the up-arrow key:

```
// register key ups
public function keyUpFunction(event:KeyboardEvent) {
    if (event.keyCode == 37) {
        leftArrow = false;
    } else if (event.keyCode == 39) {
        rightArrow = false;
    } else if (event.keyCode == 38) {
        upArrow = false;
        // remove thruster
        if (gameMode == "play") ship.gotoAndStop(1);
    }
}
```

Ship Movement

All the animation functions in this game will accept `timeDiff` as a parameter. This is just like the `timePassed` variable in other games with animation. However, instead of each animation function calculating its own `timePassed`, we calculate it in a single function, `moveGameObjects`, that then calls all three animation functions and passes it along. All the objects then move through time in step with each other.

Ship movement can mean turning, flying, or both. If the left or right arrow is pressed, the ship turns, depending on the `timeDiff` and the `shipRotationSpeed` constant.

If the up arrow is pressed, the ship should accelerate. This is where we use `Math.cos` and `Math.sin` to determine how much influence the thrust has on the horizontal and vertical movement of the ship:

```
// animate ship
public function moveShip(timeDiff:uint) {

    // rotate and thrust
    if (leftArrow) {
        ship.rotation -= shipRotationSpeed*timeDiff;
    } else if (rightArrow) {
        ship.rotation += shipRotationSpeed*timeDiff;
    } else if (upArrow) {
        shipMoveX += Math.cos(Math.PI*ship.rotation/180)*thrustPower;
        shipMoveY += Math.sin(Math.PI*ship.rotation/180)*thrustPower;
    }
```

Next, the ship's position is updated according to the velocity:

```
    // move
    ship.x += shipMoveX;
    ship.y += shipMoveY;
```

One of the things that makes this genre of games special is the way the ship can go off the screen on one side and show up on the other. Here is the code that does that. There are a lot of hard-coded numbers here that could be moved to constants at the top of the script. But, leaving them here actually makes the code easier to read and understand.

The screen is 550 pixels wide and 400 pixels high. We don't want to wrap the ship as soon as it hits the edge of the screen, but instead as it is just out of sight. So, at 570, the ship wraps back 590, putting it at –20. Because the ship would be moving to the right to do this, it will not be out of view for any period of time.

NOTE

The extra 20 pixels that we are adding at the edges of the screen is a sort of dead zone for the game. You can't see things there, and the missiles won't be there either because they die at the very edge of the screen.

You need to ensure that this area is not any larger; otherwise, small rocks moving very vertically or horizontally will get lost for a while. It would also be easy to lose your ship there if the area is too big.

But, if you make it too small, objects will seem to snap out of existence at one edge of the screen and then reappear at the other edge.

```
        // wrap around screen
        if ((shipMoveX > 0) && (ship.x > 570)) {
            ship.x -= 590;
        }
        if ((shipMoveX < 0) && (ship.x < -20)) {
            ship.x += 590;
        }
        if ((shipMoveY > 0) && (ship.y > 420)) {
            ship.y -= 440;
        }
        if ((shipMoveY < 0) && (ship.y < -20)) {
            ship.y += 440;
        }
    }
```

Handling Ship Collisions

When the ship is hit by a missile, it should explode. To do this, the ship goes to the third frame, labeled `"explode"`. The `removeAllShieldIcons` function gets rid of the shield icons on the screen. Then, a timer is set up to call `newShip` after two seconds. The number of ships is reduced by one, and the `removeShipIcon` is called to take one of the icons off the screen:

```
    // remove ship
    public function shipHit() {
        gameMode = "delay";
        ship.gotoAndPlay("explode");
        removeAllShieldIcons();
        delayTimer = new Timer(2000,1);
        delayTimer.addEventListener(TimerEvent.TIMER_COMPLETE,newShip);
        delayTimer.start();
        removeShipIcon();
        shipsLeft--;
    }
```

Shields Up!

A somewhat separate part of the ship is the shield. It exists as a movie clip inside of the ship movie clip. So, to turn it on, we just need to set its `visible` property to `true`. A timer is set to turn off the shield in three seconds. In the meantime, `shieldOn` will be set to `true`, so any passing rock collisions will be ignored.

NOTE

The shield is actually a semitransparent graphic that allows the ship to be seen through the shield. It has Alpha settings applied to the colors used in the gradient of the shield. No ActionScript is needed for this; the graphic is just drawn this way.

The `startShield` function also does some checking at the start and the end of the function. At the beginning, it makes sure the player has some shields left. Then, it makes sure the shield isn't already on.

At the end, it checks the `freeShield` parameter. If `false`, we reduce the number of available shields by one and update the screen:

```
// turn on shield for 3 seconds
public function startShield(freeShield:Boolean) {
     if (shieldsLeft < 1) return; // no shields left
     if (shieldOn) return; // shield already on

     // turn on shield and set timer to turn off
     ship.shield.visible = true;
     shieldTimer = new Timer(3000,1);
     shieldTimer.addEventListener(TimerEvent.TIMER_COMPLETE,endShield);
     shieldTimer.start();

     // update shields remaining
     if (!freeShield) {
          removeShieldIcon();
          shieldsLeft--;
     }
     shieldOn = true;
}
```

When the timer goes off, the shield is set back to `invisible`, and the `shieldOne` Boolean is set to `false`:

```
// turn off shield
public function endShield(event:TimerEvent) {
     ship.shield.visible = false;
     shieldOn = false;
}
```

Rocks

Next come the functions to handle the rocks. We have functions to create rocks, remove them, and destroy them.

Creating New Rocks

Rocks come in three sizes, so when `newRock` is called, it is with the parameter `rockType` to specify the size of the new rock. At the start of the game, all the rocks are created with `"Big"` as the size option. But, later in the game, we'll be creating pairs of rocks with every missile strike that use `"Medium"` and `"Small"` as the size.

For each size, we also have a corresponding `rockRadius` of 35, 20, and 10. We'll be using those numbers to detect collisions later on.

NOTE

It would be nice to get the radius numbers for each rock dynamically, by actually checking the rock movie clips. But at this point, we haven't created any yet, and so we can't get those values. But more important, we don't really want those values. They would include the farthest points in the graphics. We want a more modest number that gives a better approximation of the general radius of the rocks.

To finish creating the rock, a random velocity is picked, by getting random values for `dx` and `dy`. W also get a random value for `dr`, the rotation speed.

Another random element is the rock variation. Each movie clip has three frames, each with a different-looking rock.

The `rocks` array is made up of data objects that include a reference to the rock movie clip, the `dx`, `dy`, and `dr` values, the `rockType` (size) and the `rockRadius`:

```
// create a single rock of a specific size
public function newRock(x,y:int, rockType:String) {

    // create appropriate new class
    var newRock:MovieClip;
    var rockRadius:Number;
    if (rockType == "Big") {
        newRock = new Rock_Big();
        rockRadius = 35;
    } else if (rockType == "Medium") {
        newRock = new Rock_Medium();
        rockRadius = 20;
    } else if (rockType == "Small") {
        newRock = new Rock_Small();
        rockRadius = 10;
    }

    // choose a random look
    newRock.gotoAndStop(Math.ceil(Math.random()*3));

    // set start position
    newRock.x = x;
    newRock.y = y;

    // set random movement and rotation
    var dx:Number = Math.random()*2.0-1.0;
    var dy:Number = Math.random()*2.0-1.0;
    var dr:Number = Math.random();

    // add to stage and to rocks list
    gameObjects.addChild(newRock);
    rocks.push({rock:newRock, dx:dx, dy:dy, dr:dr, rockType:rockType, rockRadius:
rockRadius});
}
```

Creating Waves of Rocks

At the start of the game, and with every new wave of rocks, the following function is called to create four big rocks, all spaced evenly on the screen. Figure 7.12 shows the positions at the exact start of the game.

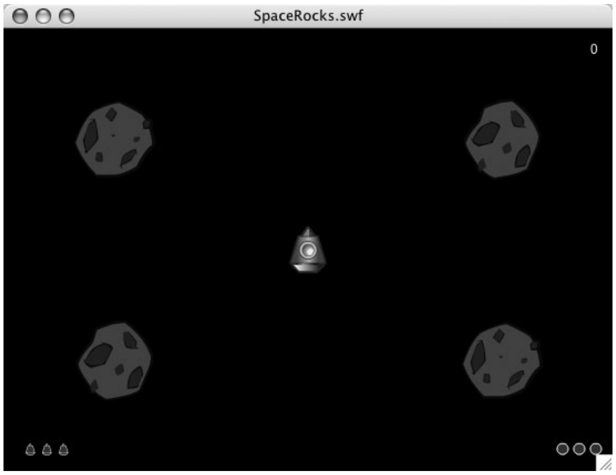


Figure 7.12
The four rocks are placed 100 pixels from the sides and top and bottom of the screen.

We want to set `gameMode` to `play`. If this is the first wave, we've already set the `gameMode` to `play`. But if this is not the first wave, then the `gameMode` would have been set to `delay` in the `shipHit` function. So we set it to `play` here to be sure:

```
// create four rocks
public function nextRockWave(event:TimerEvent) {
    rocks = new Array();
    newRock(100,100,"Big");
    newRock(100,300,"Big");
    newRock(450,100,"Big");
    newRock(450,300,"Big");
    gameMode = "play";
}
```

NOTE

The `newRockWave` function creates four rocks in the same place each time. You might want to complicate this function by checking the `gameLevel` and perhaps using a six-rock formation if the level is higher than three or four. That's an easy way to add some depth to the game. There's also no reason that some medium and small rocks can't be placed at the start of a level.

Moving Rocks

To move the rocks, we just need to look at each rock and get the values in each object of the `rocks` array. The position is changed according to the `dx` and `dy` values. The `rotation` is changed according to the `dr` value.

As with the ship, we need to wrap the rocks from one side of the screen to the other. The code to do this is almost the same, with the rocks being allowed to go 20 pixels outside of the screen before wrapping back the width of the screen plus 40 pixels (20 for each side):

```
// animate all rocks
public function moveRocks(timeDiff:uint) {
     for(var i:int=rocks.length-1;i>=0;i--) {

          // move the rocks
          var rockSpeed:Number = rockSpeedStart + rockSpeedIncrease*gameLevel;
          rocks[i].rock.x += rocks[i].dx*timeDiff*rockSpeed;
          rocks[i].rock.y += rocks[i].dy*timeDiff*rockSpeed;

          // rotate rocks
          rocks[i].rock.rotation += rocks[i].dr*timeDiff*rockSpeed;

          // wrap rocks
          if ((rocks[i].dx > 0) && (rocks[i].rock.x > 570)) {
               rocks[i].rock.x -= 590;
          }
          if ((rocks[i].dx < 0) && (rocks[i].rock.x < -20)) {
               rocks[i].rock.x += 590;
          }
          if ((rocks[i].dy > 0) && (rocks[i].rock.y > 420)) {
               rocks[i].rock.y -= 440;
          }
          if ((rocks[i].dy < 0) && (rocks[i].rock.y < -20)) {
               rocks[i].rock.y += 440;
          }
     }
}
```

Rock Collisions

When a rock is hit, the `rockHit` function decides what to do with it. In the case of a big rock, two medium rocks are created in its place. In the case of a medium rock, two small rocks are created. They start in the same location as the old rock, but get new random directions and spins.

In either case, and if it is a small rock that is hit, the original rock is removed:

```
public function rockHit(rockNum:uint) {
    // create two smaller rocks
    if (rocks[rockNum].rockType == "Big") {
        newRock(rocks[rockNum].rock.x,rocks[rockNum].rock.y,"Medium");
        newRock(rocks[rockNum].rock.x,rocks[rockNum].rock.y,"Medium");
    } else if (rocks[rockNum].rockType == "Medium") {
        newRock(rocks[rockNum].rock.x,rocks[rockNum].rock.y,"Small");
        newRock(rocks[rockNum].rock.x,rocks[rockNum].rock.y,"Small");
    }
    // remove original rock
    gameObjects.removeChild(rocks[rockNum].rock);
    rocks.splice(rockNum,1);
}
```

Missiles

Missiles are created when the player presses the spacebar. The `newMissile` function uses the position of the ship to start the missile, and also takes the rotation of the ship to determine the direction of the missile.

The placement of the missile isn't at the center of the ship, however; it is one `shipRadius` away from the center, using the same direction that the missile will continue to travel. This prevents the missiles from appearing as if they originate from the center of the ship.

NOTE

A visual trick we are using here to simplify the missiles is to have the missile graphic be a round ball. This way, we don't need to rotate the missile in any specific angle. These round objects look just fine moving in any direction.

We keep track of all missiles with the `missiles` array:

```
// create a new missile
public function newMissile() {
    // create
    var newMissile:Missile = new Missile();

    // set direction
    newMissile.dx = Math.cos(Math.PI*ship.rotation/180);
    newMissile.dy = Math.sin(Math.PI*ship.rotation/180);

    // placement
    newMissile.x = ship.x + newMissile.dx*shipRadius;
    newMissile.y = ship.y + newMissile.dy*shipRadius;

    // add to stage and array
    gameObjects.addChild(newMissile);
    missiles.push(newMissile);
}
```

As the missiles move, we use the `missileSpeed` constant and the `timeDiff` to determine the new location.

The missiles won't wrap around the screen as the rocks and ship do, but instead simply terminate when they go offscreen:

```
// animate missiles
public function moveMissiles(timeDiff:uint) {
    for(var i:int=missiles.length-1;i>=0;i--) {
        // move
        missiles[i].x += missiles[i].dx*missileSpeed*timeDiff;
        missiles[i].y += missiles[i].dy*missileSpeed*timeDiff;
        // moved off screen
        if ((missiles[i].x < 0) || (missiles[i].x > 550) ||
                (missiles[i].y < 0) || (missiles[i].y > 400)) {
            gameObjects.removeChild(missiles[i]);
            missiles.splice(i,1);
        }
    }
}
```

When a missile hits a rock, it is also taken away with a call to `missileHit`:

```
// remove a missile
public function missileHit(missileNum:uint) {
    gameObjects.removeChild(missiles[missileNum]);
    missiles.splice(missileNum,1);
}
```

NOTE

The reason we remove the missiles in `moveMissiles` with separate code instead of calling `missileHit` is just a consideration for the future. They are both happening under different circumstances. If we want something special to happen when the missile hits a target, we put it in `missileHit`. But, we probably wouldn't want that to happen if the missile just ran offscreen.

Game Control

So far, we've had three animation functions: `moveShip`, `moveRocks`, and `moveMissiles`. All three of these are called by the primary animation function, `moveGameObjects`. In turn, it is called by the `ENTER_FRAME` event we set up earlier.

Moving Game Objects

The `moveGameObjects` function calculates the `timePassed` like Air Raid did, and then sends it to all three functions. Note that `moveShip` is only called if the `gameMode` is not `"delay"`.

Finally, `moveGameObjects` calls `checkCollisions`, which is the heart of the entire game:

```
public function moveGameObjects(event:Event) {
     // get timer difference and animate
     var timePassed:uint = getTimer() - lastTime;
     lastTime += timePassed;
     moveRocks(timePassed);
     if (gameMode != "delay") {
          moveShip(timePassed);
     }
     moveMissiles(timePassed);
     checkCollisions();
}
```

Checking for Collisions

The `checkCollisions` function does the critical calculations. It loops through the rocks and the missiles and checks for any that have collided with each other. The `rockRadius` of the rocks is used to determine collisions. It is faster than calling `hitTestPoint`.

If there is a collision, the `rockHit` and `missileHit` functions are called to take care of both ends of the collision.

If a `rock` and a `missile` are to be removed at this point, it is important that neither be looked at any more for possible collisions with other objects. So, each of the two nested `for` loops have been given a label. A label is a way to specify which of the `for` loops a `break` or `continue` command is meant for. In this case, we want to continue in the `rockloop`, which is the outer of the nested loops. A simple `break` would mean that the code would continue on to check the rock against a ship collision. But, because the rock no longer exists, an error would occur:

```
// look for missiles colliding with rocks
public function checkCollisions() {
     // loop through rocks
     rockloop: for(var j:int=rocks.length-1;j>=0;j--) {
          // loop through missiles
          missileloop: for(var i:int=missiles.length-1;i>=0;i--) {
               // collision detection
               if (Point.distance(new Point(rocks[j].rock.x,rocks[j].rock.y),
                         new Point(missiles[i].x,missiles[i].y))
                              < rocks[j].rockRadius) {

                    // remove rock and missile
                    rockHit(j);
                    missileHit(i);

                    // add score
                    gameScore += 10;
                    updateScore();

                    // break out of this loop and continue next one
                    continue rockloop;
                }
            }
```

Each rock is checked to see whether it collides with the ship. First, we need to make sure we aren't in the time between the ship's demise and its regeneration. We also need to make sure the shield is down.

If the ship is hit, `shipHit` and `rockHit` are both called:

```
            // check for rock hitting ship
            if (gameMode == "play") {
                if (shieldOn == false) { // only if shield is off
                    if (Point.distance(new Point(rocks[j].rock.x,rocks[j].rock.y),
                            new Point(ship.x,ship.y))
                                < rocks[j].rockRadius+shipRadius) {

                        // remove ship and rock
                        shipHit();
                        rockHit(j);
                    }
                }
            }
        }
```

Before `checkCollisions` is done, it takes a quick look at the number of rocks on the screen. If all have been wiped out, a timer is set up to start a new set in two seconds. The `gameLevel` is upped by one, so the next rocks will be a bit faster. Also, the `gameMode` is set to `"betweenlevels"`. This means the check won't be performed again until the rocks reappear, but it still allows for ship movement by the player:

```
        // all out of rocks, change game mode and trigger more
        if ((rocks.length == 0) && (gameMode == "play")) {
            gameMode = "betweenlevels";
            gameLevel++; // advance a level
            delayTimer = new Timer(2000,1);
            delayTimer.addEventListener(TimerEvent.TIMER_COMPLETE,nextRockWave);
            delayTimer.start();
        }
    }
```

Ending the Game

If the ship has been hit, and there are no more ships remaining, the game is over, and `endGame` is called. It does the typical cleanup and sends the movie to the third frame on the timeline:

```
public function endGame() {
    // remove all objects and listeners
    removeChild(gameObjects);
    removeChild(scoreObjects);
    gameObjects = null;
    scoreObjects = null;
    removeEventListener(Event.ENTER_FRAME,moveGameObjects);
    stage.removeEventListener(KeyboardEvent.KEY_DOWN,keyDownFunction);
    stage.removeEventListener(KeyboardEvent.KEY_UP,keyUpFunction);

    gotoAndStop("gameover");
}
```

Modifying the Game

The shield feature in this game is actually not in the original Asteroids game. But, it can be found in sequels and many other games of the genre. What is in the original is a "warp" feature. This is where the ship disappears and reappears at a random location. Although this often leads to doom, it is a good last resort for players when they can't escape a tight spot.

Adding a warp feature is as easy as accepting a new key press in the `keyDownFunction` and then assigning a random `x` and `y` value to the ship.

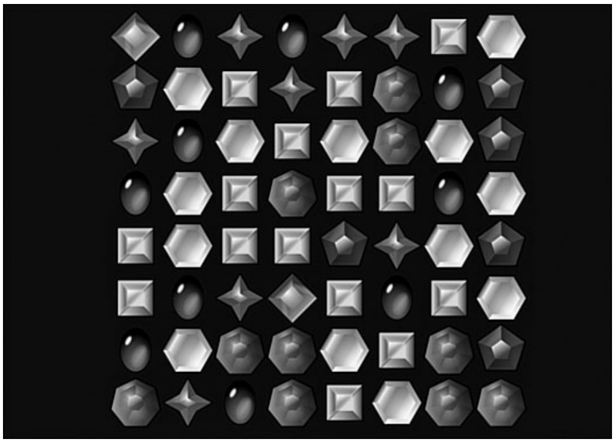
This game can benefit from some basics: like sound, or more animation. The thruster flame can be animated by simply replacing the flame graphic with a looping graphic symbol in the movie clip. No ActionScript necessary.

You could also add bonus lives, a common feature in these sorts of games. Just look for key scoring goals, such as 1,000, and add to `shipsLeft`. You'll want to redraw the ship icons at that time, and perhaps play a sound to indicate the bonus.

Most games of this genre on the Web aren't space games at all. This general concept can be used in education or marketing by replacing the rocks with specific objects. For instance, they could be nouns and verbs, and the player is supposed to only shoot the nouns. Or they could be pieces of trash that you are supposed to be cleaning up.

A simple modification would be to forget about the missiles completely and make collisions with the rocks and ship desirable. You could be gathering objects, in that case, rather than shooting them. But, perhaps you want to gather some objects, and you want to avoid others.

This page intentionally left blank



8

Casual Games: Match Three

- Reusable Class: Point Bursts
- Match Three

In their beginning, video games were simple and fun. Little-action puzzle games such as Tetris were the most popular. Then, 3D graphics pushed the edge of gaming into the virtual worlds of first-person shooters and online role-playing games.

However, puzzle games came back into popularity in the early part of this decade as online free and downloadable games. These are usually called *casual games.*

NOTE

There is a lot of confusion over the term *casual game.* Wikipedia defines it as "a category of electronic or computer games targeted at a mass audience." This is a pretty broad definition. A narrow one is simply, "Match Three games," because most websites that sell "casual games" are selling mostly Match Three games.

However, many of the games in this book fit the wider definition. In fact, many picture-puzzle and word-puzzle games are sold alongside Match Three.

Most casual games are action puzzle games, meaning they combine a puzzle game with some sort of movement or a time limit to elevate the level of excitement.

The Match Three game is by far the most common of all casual games. Probably about half the games at popular casual game websites are some form of Match Three.

This chapter takes a look at point bursts, a popular special effect used in casual games. Then, we go on to build a typical Match Three game.

Reusable Class: Point Bursts

SOURCE FILES
http://flashgameu.com
A3GPU08_PointBurst.zip

In the old days of arcade games, when you did something right, you were awarded points. That hasn't changed. But, what has changed is the standard way of indicating it.

In the old arcade games, you would simply see your score change in a corner of the screen. Chances are you weren't watching this corner of the screen at the time, and wouldn't look at it until the action was over. So, it makes sense that games evolved to show you how many points you received right at the location of the screen where your action took place.

Check almost any well-built casual game and you'll see this. Figure 8.1 shows my game Gold Strike right at the moment that the player clicks some gold blocks to score points. You can see the "30" text in the location where the gold blocks used to be. These numbers grow from small to large in an instant and then disappear. They are there just long enough to show players how many points they have scored.

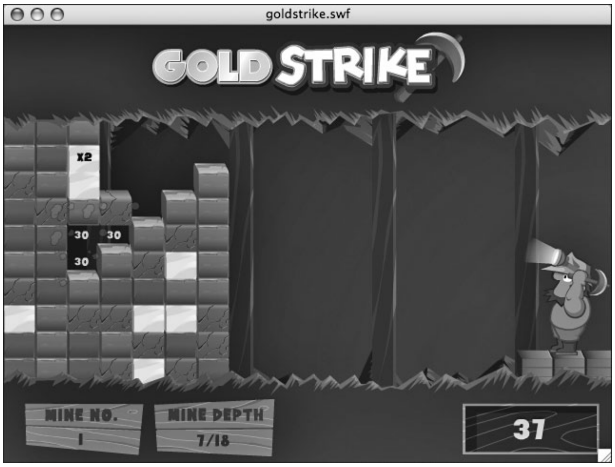


Figure 8.1
The number of points scored shows up right at the spot where the action occurred.

I call this special effect a *point burst*. It is so common, and I use it so frequently, that it is an ideal candidate for a special class that can be built and then reused in many games.

Developing the Point Burst Class

The **`PointBurst.as`** class should be as self-contained as possible. In fact, our goal is to be able to use a point burst with only one line of code in the game. So, the class itself needs to take care of creating the text and sprite, animating it, and removing itself completely when done.

NOTE
Not only will our `PointBurst` class need just one line of code to use, but it will not require any items in the main movie's library other than a font to use in the point burst.

Figure 8.2 shows a time-lapse version of what we are going for. The point burst should start small, and then grow in size. It should also start at 100 percent opacity and fade away to become transparent. And, it should do this in less than a second.

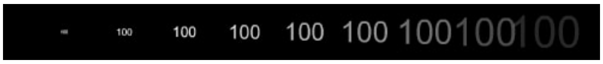


Figure 8.2
This time-lapse image shows the start of the point burst at the left, and then each stage of the animation from left to right.

The Class Definition

For such a small class, we still need four imports. We'll be using the timer to control the animation of the point burst, although another option is to make it time based using `ENTER_FRAME` events:

```
package {
     import flash.display.*;
     import flash.events.*;
     import flash.text.*;
     import flash.utils.Timer;
```

Even though the `PointBurst` class is performing animation, it is still a sprite, because it doesn't require multiple frames. Instead, we'll be scaling and setting the `alpha` of the sprite in each time step.

We will use static constants to decide the font type, size, and color:

```
     public class PointBurst extends sprite {
          // text style
          static const fontFace:String = "Arial";
          static const fontSize:int = 20;
          static const fontBold:Boolean = true;
          static const fontColor:Number = 0xFFFFFF;
```

We also have several constants associated with the animation. The `animSteps` and `animStepTime` determine the length and smoothness of the animation. For instance, at 10 steps, with 50 milliseconds between steps, it takes 500 milliseconds to animate; 20 steps at 25 milliseconds between steps also takes 500 milliseconds, but includes twice as many steps for smoother animation:

```
          // animation
          static const animSteps:int = 10;
          static const animStepTime:int = 50;
```

The scale of the movie changes during the animation. These two constants set the starting point and end point of the change in scale:

```
          static const startScale:Number = 0;
          static const endScale:Number = 2.0;
```

After the constants, we have several variables to hold references to the items in the point burst. One holds the text field, and another the `Sprite` that will encapsulate the text field. A third holds a reference to the stage or movie clip where we want to place the point burst. The last holds a reference to the `Timer` object:

```
          private var tField:TextField;
          private var burstSprite:Sprite;
          private var parentMC:MovieClip;
          private var animTimer:Timer;
```

The PointBurst Function

The one line of code we use to create a `PointBurst` is to create a new `PointBurst` object. This in turn calls the `PointBurst` function, which accepts parameters. These parameters are our only way to communicate to the `PointBurst` object some key information, such as the location of the point burst and what text to display.

NOTE

The `pts` parameter is an `Object` because we want to be able to accept any kind of variable type: `int`, `Number`, or `String`. We'll convert whatever it is to a `String` later, because that is what the `text` property of a `TextField` requires.

The first parameter of `PointBurst` is a movie clip, `mc`. This will be a reference to the stage or another movie clip or sprite where the point burst will be added with `addChild`:

```
public function PointBurst(mc:MovieClip, pts:Object, x,y:Number) {
```

The first thing the function must do is to create a `TextFormat` object to assign to the `TextField` we'll create later. This will include use of the formatting constants we defined earlier. It will also set the alignment of the field to `"center"`:

```
    // create text format
    var tFormat:TextFormat = new TextFormat();
    tFormat.font = fontFace;
    tFormat.size = fontSize;
    tFormat.bold = fontBold;
    tFormat.color = fontColor;
    tFormat.align = "center";
```

Next, we create the `TextField` itself. In addition to turning `selectable` to `false`, we also need to tell the field to use embedded fonts rather than system fonts. This is because we want to set the transparency of the text, something that can only be done when the text uses embedded fonts.

To get the text to be centered in the sprite we'll create next, we set the `autoSize` of the field to `TextFieldAutoSize.CENTER`. Then, we set the `x` and `y` properties to negative half of the `width` and `height`. This puts the center of the text at point 0,0:

```
    // create text field
    tField = new TextField();
    tField.embedFonts = true;
    tField.selectable = false;
    tField.defaultTextFormat = tFormat;
    tField.autoSize = TextFieldAutoSize.CENTER;
    tField.text = String(pts);
    tField.x = -(tField.width/2);
    tField.y = -(tField.height/2);
```

Now we create a sprite to hold the text and act as the main display object for the animation. We set the location of this sprite to the `x` and `y` values passed into the function. We set the scale of the sprite to the `startScale` constant. We set the `alpha` to zero. Then, we add the sprite to the `mc` movie clip, which is the sprite passed in to the function:

```
        // create sprite
        burstSprite = new Sprite();
        burstSprite.x = x;
        burstSprite.y = y;
        burstSprite.scaleX = startScale;
        burstSprite.scaleY = startScale;
        burstSprite.alpha = 0;
        burstSprite.addChild(tField);
        parentMC = mc;
        parentMC.addChild(burstSprite);
```

Now that the `PointBurst` object has manifested itself as a sprite, we just need to start a timer to control the animation over the next 500 milliseconds. This timer calls `rescaleBurst` several times, and then calls `removeBurst` when it is done:

```
        // start animation
        animTimer = new Timer(animStepTime,animSteps);
        animTimer.addEventListener(TimerEvent.TIMER, rescaleBurst);
        animTimer.addEventListener(TimerEvent.TIMER_COMPLETE, removeBurst);
        animTimer.start();
    }
```

Animating the Point Burst

When the `Timer` calls `rescaleBurst`, we need to set the scale properties and the `alpha` of the sprite. First, we calculate `percentDone` based on how many `Timer` steps have gone by and the total number of `animSteps`. Then, we apply this value to the `startScale` and `endScale` constants to get the current scale. We can use `percentDone` to set the `alpha`, but we want to invert the value so that the `alpha` goes from 1.0 to 0.0.

NOTE

The `alpha` property sets the transparency of a sprite or movie clip. At 1.0, the object behaves as normal, filling in solid colors at 100 percent opacity. This still means that unfilled areas, like those outside the shape of the characters, are transparent. At .5, or 50 percent transparency, the areas that are usually opaque, like the lines and fills of the characters, share the pixels with the colors behind them.

```
    // animate
    public function rescaleBurst(event:TimerEvent) {
        // how far along are we
        var percentDone:Number = event.target.currentCount/animSteps;
        // set scale and alpha
        burstSprite.scaleX = (1.0-percentDone)*startScale + percentDone*endScale;
        burstSprite.scaleY = (1.0-percentDone)*startScale + percentDone*endScale;
        burstSprite.alpha = 1.0-percentDone;
    }
```

When the `Timer` is done, it will call `removeBurst`. This takes care of everything needed for the `PointBurst` to get rid of itself, without any action on the part of the main movie or the movie's class.

After removing the `tField` from the `burstSprite`, the `burstSprite` is removed from the `parentMC`. Then, both are set to `null` to clear them from memory. Finally, `delete` is used to clear the `PointBurst` object away completely.

NOTE

It is unclear whether you need all the lines in `removeBurst`. You are supposed to clear away all references to an object to delete it. But, the `delete` statement removes the `PointBurst`, which in turn should remove the two variables, too. Removing the `burstSprite` may also serve to remove the `tField`. There is no way to test this, and at the time of this writing, there doesn't seem to be any technical document that tells us what the Flash player does in this case, specifically. So, it is best to use a function that ensures all of this is cleared.

```
    // all done, remove self
    public function removeBurst(event:TimerEvent) {
        burstSprite.removeChild(tField);
        parentMC.removeChild(burstSprite);
        tField = null;
        burstSprite = null;
        delete this;
    }
```

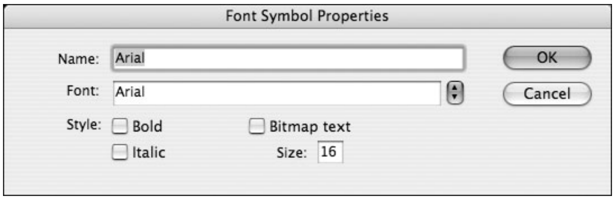
Using Point Bursts in a Movie

You need to do two things before creating a new `PointBurst` object in a movie. The first is to create a `Font` object in the movie's library. The second is to tell Flash where to look to find your **PointBurst.as** file.

Adding a Font to a Movie

The reason a `Font` is needed is because we are using `alpha` to adjust the transparency of the text. This can only be done with an embedded `Font` in the library.

To create an embedded `Font`, you need to use the Library panel's drop-down menu and choose New Font. Then, select the font you want and name it. The name of the library element doesn't really matter, so I usually use the font's name, like Arial for this. Figure 8.3 shows the Font Symbol Properties dialog box.

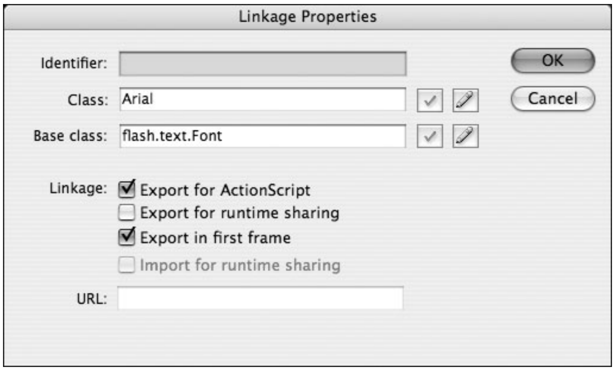


Font Symbol Properties

Name: Arial
Font: Arial
Style: [ ] Bold [ ] Italic [ ] Bitmap text Size: 16

Figure 8.3
The Font Symbol Properties lets you choose a font to add to the library.

But, this is only step one. Step two, which is not obvious at all, is to make sure this font is included for ActionScript use. To do this, select the font in the library. Then, right-click in Windows or Control-click on Mac and select the Linkage option. This brings up the Linkage Properties dialog shown in Figure 8.4.



Linkage Properties

Identifier:
Class: Arial
Base class: flash.text.Font
Linkage: [✓] Export for ActionScript
[ ] Export for runtime sharing
[✓] Export in first frame
[ ] Import for runtime sharing
URL:

Figure 8.4
Within the Linkage Properties dialog, you can specify a class for a font in the library.

NOTE
Unlike other symbol types, the linkage options do not show up in the general symbol Properties dialog. They only show in the Linkage Properties dialog.

It doesn't matter what class name you give to the font. And, you can ignore the dialog that tells you that there is no class definition found. None is needed.

Class Locations

For our examples, we don't need to do anything to tell Flash where to look to find out **PointBurst.as** class file. This is because it is in the same location as the Flash movie.

But, if you want to use the same **PointBurst.as** class file in multiple projects, you need to locate it somewhere where all the project movies can get to it, and then tell them where to find it.

There are two ways to do this. The first is to add a class path to the Flash preferences. You might want to create a folder to hold all the classes you regularly use. Then, go to the Flash Preferences, ActionScript section. There, you can click the ActionScript 3.0 Settings button and add a folder to the place where Flash looks for class files.

NOTE

Alternatively, you could just use the default location for library classes, the Flash Classes folder, which is in your Flash folder in the Program Files or Applications folder. I don't like doing this because I try to keep any of the documents I create out of the Applications folder, leaving only the default install of my applications there.

A second way to tell a movie to find a class file not in the same directory as the movie is to go to File, Publish Settings and click the Settings button next to the ActionScript version selection. Then, you can add a new class path for only this one movie.

To summarize, here are the four ways a Flash movie can access a class file:

1. Place the class file in the same folder as the movie.
2. Add the class location in the Flash Preferences.
3. Place the class file in the Flash application Class folder.
4. Add the class location in the movie's Publish Settings.

Creating a Point Burst

After you have the font in the library, and the movie has access to the class, it just takes one line to make a point burst. Here is an example:

```
var pb:PointBurst = new PointBurst(this,100,50,75);
```

This creates a point burst with the number 100 displayed. The burst will appear at location 50,75.

The example movie **PointBurstExample.fla** and its accompanying **PointBurstExample.as** class present a slightly more advanced example. It creates a point burst wherever you click:

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

    public class PointBurstExample extends MovieClip {

        public function PointBurstExample() {
            stage.addEventListener(MouseEvent.CLICK,tryPointBurst);
        }

        public function tryPointBurst(event:MouseEvent) {
            var pb:PointBurst = new PointBurst(this,100,mouseX,mouseY);
        }
    }
}
```

Now that we have an independent piece of code that takes care of this somewhat complex special effect, we can move on to our next game knowing that it can include the point burst with almost no additional programming effort.

Match Three

SOURCE FILES
http://flashgameu.com
A3GPU08_MatchThree.zip

Although Match Three is the most common and popular casual game, it didn't get that way because it was easy to program. In fact, many aspects of Match Three require some very tricky techniques. We'll look at the game piece by piece.

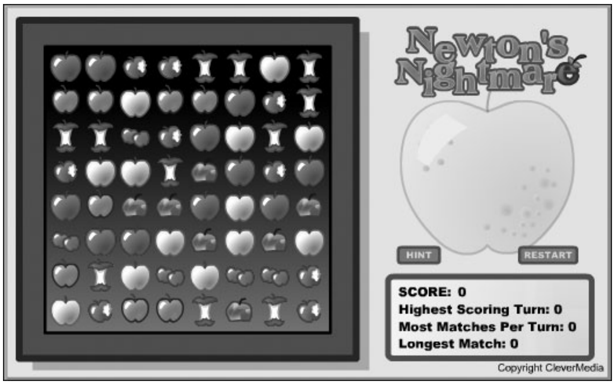
Playing Match Three

In case you have been successful in avoiding Match Three games over the past few years, here is how they are played.

An eight-by-eight board holds a random arrangement of six or seven game pieces. You can click any two horizontally or vertically adjacent pieces two try to swap them. If the swap results in a horizontal or vertical lineup of three or more of the same types of pieces, the swap is allowed. The pieces that line up are then removed, with pieces above them dropping down. More pieces drop from above to fill the gap left by the match.

That's it. It is the simplicity of the game that is part of what makes it popular. The game continues until the board reaches a state where no more moves are possible.

Figure 8.5 shows my game Newton's Nightmare, a fairly typical Match Three game.



SCORE: 0
Highest Scoring Turn: 0
Most Matches Per Turn: 0
Longest Match: 0

Figure 8.5
Newton's Nightmare features apples as the playing pieces in a Match Three game.

NOTE
The game Bejeweled, also named Diamond Mine, is credited with kicking off the Match Three craze.

Game Functionality Overview

The sequence of events in the game follows 12 steps. Each step presents a different programming challenge.

1. Create a Random Board

An eight-by-eight board with a random arrangement of seven different items is created to start the game

2. Check for Matches

There are some restrictions on what the initial board can hold. The first is that the board can include no three-in-a-row matches. It must be up to the player to find the first match.

3. Check for Moves

The second restriction on the initial board is that there must be at least one valid move. That means the player must be able to swap two pieces and create a match.

4. Player Selects Two Pieces

The pieces must be adjacent to each other horizontally or vertically, and the swap must result in a match.

5. The Pieces Are Swapped

Usually an animation shows the two pieces moving into each others' places.

6. Look for Matches

After a swap is made, the board should be searched for new matches of three in a row or more. If no match is found, the swap needs to be reversed.

7. Award Points

If a match is found, points should be awarded.

8. Remove Matches

The pieces involved in a match should be removed from the board.

9. Drop Down

The pieces above the ones removed need to drop down to fill the space.

10. Add New

New pieces need to drop down from above the board to fill in empty spaces.

11. Look for Matches Again

After all pieces have dropped and new ones have filled in the gaps, another search for matches is needed. Back to step 6.

12. Check for No More Moves

Before giving control back to the player, a check is made to see whether any moves are possible at all. If not, the game is over.

The Movie and `MatchThree` Class

The **`MatchThree.fla`** movie is pretty simple. Besides the Arial font in the library, the only game-related elements are a movie clip for the game pieces, and another clip that acts as a selection indicator.

Figure 8.6 shows the `Piece` movie clip. There are seven frames, each with a different piece. There is also the select movie clip on the top layer, across all seven frames. This can be turned on or off using the `visible` property.

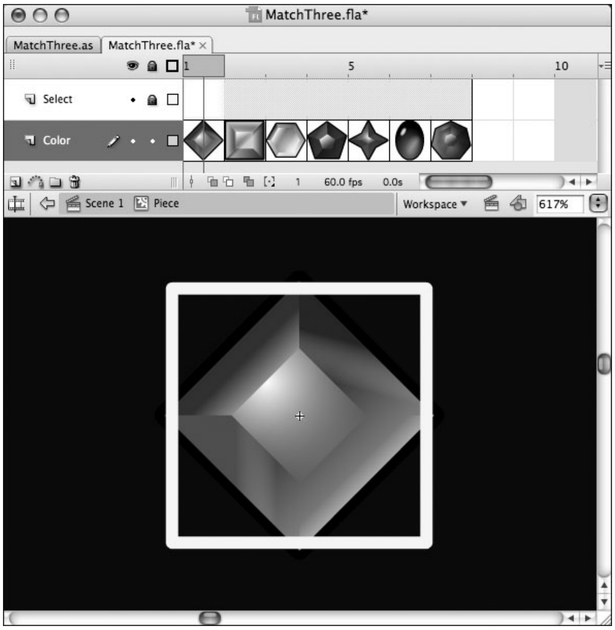


Figure 8.6
The `Piece` movie clip contains seven variations and a selection box.

Let's get the class definitions out of the way before looking at the game logic. Surprisingly, there isn't too much to define. Only the most basic imports are needed:

```
package {
     import flash.display.*;
     import flash.events.*;
     import flash.text.*;
     import flash.utils.Timer;
```

As for constants, we just have some for the number of variations for the `Piece`, and three constants that have to do with screen display position:

```
     public class MatchThree extends MovieClip {
          // constants
          static const numPieces:uint = 7;
          static const spacing:Number = 45;
          static const offsetX:Number = 120;
          static const offsetY:Number = 30;
```

The game state will be stored in five different variables. The first, `grid`, contains references to all the `Pieces`. It is actually an array of arrays. So, each item in `grid` is actually another array containing eight `Piece` movie clip references. So, itis an eight-by-eight nested array. Then, we can look at any `Piece` by simply using `grid[x][y]`.

The `gameSprite` is a sprite that holds all the sprites and movie clips we'll be creating. This keeps them separate from any other graphics already on the stage.

The `firstPiece` variable holds a reference to the first `Piece` clicked, much like the matching game did in Chapter 3, "Basic Game Framework: A Matching Game."

The two Boolean variables, `isDropping` and `isSwapping`, keep track of whether any `Pieces` are animating at the moment. The `gameScore` variable holds the player's score:

```
          // game grid and mode
          private var grid:Array;
          private var gameSprite:Sprite;
          private var firstPiece:Piece;
          private var isDropping,isSwapping:Boolean;
          private var gameScore:int;
```

Setting Up the Grid

The first functions will set the game variables, including setting up the game grid.

Setting the Game Variables

To start the game, we need to set all the game state variables. We start by creating the `grid` array of arrays. Then, we call `setUpGrid` to populate it.

NOTE

There is no need to fill the internal arrays of `grid` with empty slots. Just by setting a location in an array, the slot in the array is created, and any earlier slots are filled with `undefined`. For instance, if a new array is created, and then item three is set to `"My String"`, the array will have a `length` of 3, and items 0 and 1 will have a value of `undefined`.

Then, we set the `isDropping`, `isSwapping`, and `gameScore` variables. Also, we set up an `ENTER_FRAME` listener to run all the movement in the game:

```
// set up grid and start game
public function startMatchThree() {
     // create grid array
     grid = new Array();
     for(var gridrows:int=0;gridrows<8;gridrows++) {
          grid.push(new Array());
     }
     setUpGrid();
     isDropping = false;
     isSwapping = false;
     gameScore = 0;
     addEventListener(Event.ENTER_FRAME,movePieces);
}
```

Setting Up the Grid

To set up the grid, we begin an endless loop using a `while(true)` statement. Then, we create the items in the grid. We plan on the very first attempt creating a valid board.

A new `gameSprite` is created to hold the movie clips for the game `Pieces`. Then, 64 random `Pieces` are created through the `addPiece` function. We look at this function next, but for now you should know that it will add a `Piece` to the grid array and also to the `gameSprite`:

```
public function setUpGrid() {
     // loop until valid starting grid
     while (true) {
          // create sprite
          gameSprite = new Sprite();

          // add 64 random pieces
          for(var col:int=0;col<8;col++) {
               for(var row:int=0;row<8;row++) {
                    addPiece(col,row);
               }
          }
```

Next, we've got to check two things to determine whether the grid that is created is a valid starting point. The lookForMatches function returns an array of matches found. We'll look at it later in this chapter. At this point, it needs to return zero, which means that there are no matches on the screen. A continue command skips the rest of the while loop and starts again by creating a new grid.

After that, we call lookForPossibles, which checks for any matches that are just one move away. If it returns false, this isn't a good starting point because the game is already over.

If neither of these conditions are met, the break command allows the program to leave the while loop. Then, we add the gameSprite to the stage:

```
            // try again if matches are present
            if (lookForMatches().length != 0) continue;

            // try again if no possible moves
            if (lookForPossibles() == false) continue;

            // no matches, but possibles exist: good board found
            break;
        }

        // add sprite
        addChild(gameSprite);
    }
```

Adding Game Pieces

The addPiece function creates a random Piece at a column and row location. It creates the movie clip and set its location:

```
    // create a random piece, add to sprite and grid
    public function addPiece(col,row:int):Piece {
        var newPiece:Piece = new Piece();
        newPiece.x = col*spacing+offsetX;
        newPiece.y = row*spacing+offsetY;
```

Each Piece needs to keep track of its own location of the board. Two dynamic properties, col and row, will be set to this purpose. Also, type holds the number corresponding to the type of Piece, which also corresponds to the frame in the movie clip being shown:

```
        newPiece.col = col;
        newPiece.row = row;
        newPiece.type = Math.ceil(Math.random()*7);
        newPiece.gotoAndStop(newPiece.type);
```

The select movie clip inside the `Piece` movie clip is the outline that appears when the user clicks a `Piece`. We'll set that to not be `visible` at the start. Then, the `Piece` will be added to the `gameSprite`.

To put the `Piece` into the `grid` array, we use a double-bracket method of addressing the nested array: `grid[col][row] = newPiece`.

Each `Piece` is given its own click listener. Then, the reference to the `Piece` is returned. We won't be using it in the `setUpGrid` function above, but we will be using it later on when creating new `Pieces` to replace matched ones:

```
        newPiece.select.visible = false;
        gameSprite.addChild(newPiece);
        grid[col][row] = newPiece;
        newPiece.addEventListener(MouseEvent.CLICK,clickPiece);
        return newPiece;
    }
```

Figure 8.7 shows a complete random valid grid.

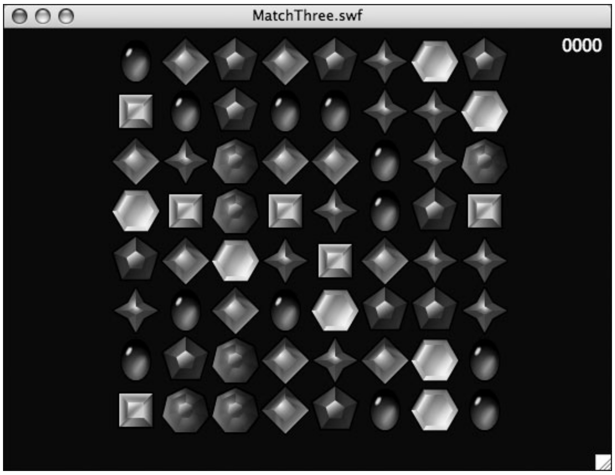


Figure 8.7
Just one of a virtually infinite number of randomly generated game grids.

Player Interaction

When the player clicks a `Piece`, what happens depends on whether it is the first `Piece` clicked, or the second. If it is the first `Piece`, the `Piece` is selected, and nothing else happens.

If the player clicks the same `Piece` twice, it is deselected and the player is back to square one:

```
// player clicks a piece
public function clickPiece(event:MouseEvent) {
    var piece:Piece = Piece(event.currentTarget);

    // first one selected
    if (firstPiece == null) {
        piece.select.visible = true;
        firstPiece = piece;

    // clicked on first piece again
    } else if (firstPiece == piece) {
        piece.select.visible = false;
        firstPiece = null;
```

If the player has clicked a second `Piece`, we need to determine whether there can be a swap. First, we turn off the selection highlight on the first `Piece`.

The first test is to determine whether the two `Pieces` are on the same row, and next to each other. Alternatively, the `Pieces` can be on the same column, and above or below the other.

In either case, `makeSwap` is called. This takes care of checking to see whether a swap is valid—that it will result in a match. If it is, or if it isn't, the `firstPiece` variable is set to `null` to get ready for the next player selection.

On the other hand, if the player has selected a `Piece` farther away from the first, it is assumed that the player wants to abandon his first selection and start selecting a second pair:

```
    // clicked second piece
    } else {
        firstPiece.select.visible = false;

        // same row, one column over
        if (firstPiece.row == piece.row) {
            if (Math.abs(firstPiece.col-piece.col) == 1) {
                makeSwap(firstPiece,piece);
                firstPiece = null;
            }

        // same column, one row over
        } else if (firstPiece.col == piece.col) {
            if (Math.abs(firstPiece.row-piece.row) == 1) {
                makeSwap(firstPiece,piece);
                firstPiece = null;
            }

        // bad move, reassign first piece
```

```
            } else {
                firstPiece = piece;
                firstPiece.select.visible = true;
            }
        }
    }
```

The `makeSwap` function swaps the two `Pieces`, and then checks to see whether a match is available. If not, it swaps the `Pieces` back. Otherwise, the `isSwapping` variable is set to `true` so that the animation can play:

```
// start animated swap of two pieces
public function makeSwap(piece1,piece2:Piece) {
    swapPieces(piece1,piece2);

    // check to see if move was fruitful
    if (lookForMatches().length == 0) {
        swapPieces(piece1,piece2);
    } else {
        isSwapping = true;
    }
}
```

To actually do the swapping, we need to store the location of the first `Piece` in a temporary variable. Then, we'll set the location of the first `Piece` to the location of the second. Figure 8.8 shows a diagram of how a swap like this works.

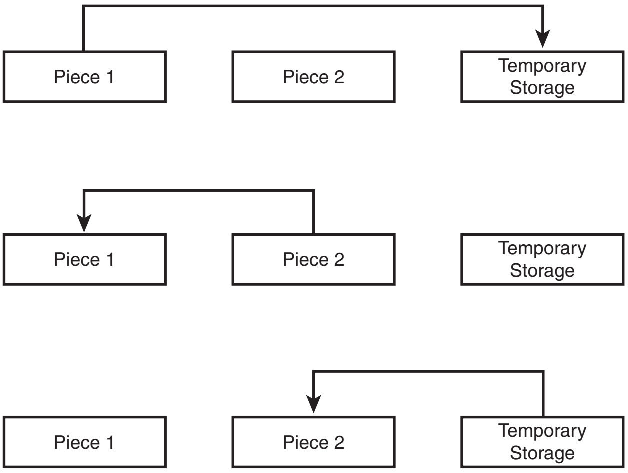


Figure 8.8
When swapping two values, it is necessary to create a temporary variable that stores one value during the swap.

When the locations of the `Pieces` are exchanged, the grid needs to be updated. Because each `Piece` now has the correct `row` and `col` value, we just set the grid to point to each `Piece` at the correct position inside of `grid`:

```
// swap two pieces
public function swapPieces(piece1,piece2:Piece) {
     // swap row and col values
     var tempCol:uint = piece1.col;
     var tempRow:uint = piece1.row;
     piece1.col = piece2.col;
     piece1.row = piece2.row;
     piece2.col = tempCol;
     piece2.row = tempRow;

     // swap grid positions
     grid[piece1.col][piece1.row] = piece1;
     grid[piece2.col][piece2.row] = piece2;
}
```

The swap is completely reversible, which is important because it will often need to be reversed. In fact, we don't know whether the swap leads to a match until after the swap is performed. So, we often need to swap the `Pieces`, check for a match, and then swap back if no match is found.

Animating Piece Movement

We're going to be using an interesting, but nonobvious, method of animating `Piece` movement. Each `Piece` knows what `row` and `col` it is in because it has a `row` and `col` dynamic property. It also knows what location it is in on the screen thanks to its `x` and `y` property.

These two should match, with help from the `spacing`, and `offsetX` and `offsetY` variables. So, a `Piece` in column 3 should be at an `x` location of `3*spacing+offsetX`.

But, what if a `Piece` moves to a new column? If we set the `col` value of the `Piece` to 4, it should be at `4*spacing+offsetX`, which is `spacing` (45) pixels to the right. In that case, we can ask the `Piece` to move a bit to the right, to get closer to its new location. If we do this each frame, the `Piece` eventually gets to its new destination, and stops moving (because it will again have a matching `col` and `x` value).

Using this technique, we can have any `Piece` animate as it moves to a new location. And we don't even need to set up this animation on a per-`Piece` level. All we need to do is change a `col` or `row` property of a `Piece`, and then the following function will take care of the rest.

The `movePieces` function is called every `ENTER_FRAME` as we set it up with a listener at the start of the class. It loops through the `Pieces` and checks all the `col` and `row` values to see whether the `x` and `y` values need adjusting to match.

NOTE

We're using a distance of 5 in `movePieces` each frame. For the `col` and `row` to line up with an `x` and `y` value, we need to stick to multiples of 5 for `spacing`. In the example movie, `spacing` is set to 45, so this works. If you were to change `spacing` to, say 48, you need to choose a new movement amount that divides evenly into 48, like 4, 6, or 8.

```
public function movePieces(event:Event) {
    var madeMove:Boolean = false;
    for(var row:int=0;row<8;row++) {
        for(var col:int=0;col<8;col++) {
            if (grid[col][row] != null) {

                // needs to move down
                if (grid[col][row].y <
                       grid[col][row].row*spacing+offsetY) {
                    grid[col][row].y += 5;
                    madeMove = true;

                // needs to move up
                } else if (grid[col][row].y >
                       grid[col][row].row*spacing+offsetY) {
                    grid[col][row].y -= 5;
                    madeMove = true;

                // needs to move right
                } else if (grid[col][row].x <
                       grid[col][row].col*spacing+offsetX) {
                    grid[col][row].x += 5;
                    madeMove = true;

                // needs to move left
                } else if (grid[col][row].x >
                       grid[col][row].col*spacing+offsetX) {
                    grid[col][row].x -= 5;
                    madeMove = true;
                }
            }
        }
    }
```

At the start of `movePieces`, we set the Boolean `madeMove` to `false`. Then, if any animation is required, we set it to `true`. In other words, if `movePieces` does nothing, `madeMove` is `false`.

Then, this value is compared to the class properties `isDropping` and `isSwapping`. If `isDropping` is `true` and `madeMove` is `false`, it must mean that all the `Pieces` that were dropping have just finished. It is time to look for more matches.

Also, if `isSwapping` is `true` and `madeMove` is `false`, it must mean that two `Pieces` just finished swapping. In this case, it is also time to look for matches:

```
     // if all dropping is done
     if (isDropping && !madeMove) {
         isDropping = false;
         findAndRemoveMatches();

     // if all swapping is done
     } else if (isSwapping && !madeMove) {
         isSwapping = false;
         findAndRemoveMatches();
     }
 }
```

Finding Matches

There are two very challenging parts to the Match Three program. The first is finding any matches in a board. This is an excellent example of the "break it into smaller problems" programming technique that I wrote about in Chapter 1, "Using Flash and ActionScript 3.0."

The problem of finding matches of three or more consecutive `Pieces` in the game grid is certainly nontrivial. It cannot be solved in a simple step. So, you cannot think of it as a single problem to solve.

Breaking the Task into Smaller Steps

Instead, we need to break it down into smaller problems, and keep breaking it down until we have simple enough problems that can be solved easily.

So, the `findAndRemoveMatches` first breaks the task into two `Pieces`: finding matches, and removing them. Removing `Pieces` is actually a pretty simple task. It just involves removing the `Piece` objects from the `gameSprite`, setting the `grid` location to `null`, and giving the player some points.

NOTE

The number of points awarded depends on the number of `Pieces` in the match. Three `Pieces` means `(3-1)*50` or 100 points per `Piece` for a total of 300 points. Four `Pieces` would be `(4-1)*50` or 150 points per `Piece` for a title of 600 points.

However, the absence of some `Pieces` means that the ones above it will need to be told they are hanging in mid air and need to drop. This is a nontrivial task, too.

So, we have two nontrivial tasks: looking for matches and telling the `Pieces` above any `Pieces` that have been removed that they need to drop. We'll delegate these two tasks to other functions: `lookForMatches` and `affectAbove`. The rest of the simple tasks we'll perform right here in the `findAndRemoveMatches` function.

The `findAndRemoveMatches` Function

We loop grab the matches found and put them into the array `matches`. Then, we award points for each match. Next, we loop through all the `Pieces` to be removed and remove them.

NOTE

When you take difficult tasks and delegate them to new functions—functions you haven't created yet—it is called *top-down programming*. Instead of worrying about how we'll find matches, we simply envision a `lookForMatches` function that will perform the task. We are building the program from the top down, taking care of the big picture first, and worrying about the functions that handle the smaller details later.

```
// gets matches and removes them, applies points
public function findAndRemoveMatches() {
     // get list of matches
     var matches:Array = lookForMatches();
     for(var i:int=0;i<matches.length;i++) {
          var numPoints:Number = (matches[i].length-1)*50;
          for(var j:int=0;j<matches[i].length;j++) {
               if (gameSprite.contains(matches[i][j])) {
                    var pb = new
PointBurst(this,numPoints,matches[i][j].x,matches[i][j].y);
                    addScore(numPoints);
                    gameSprite.removeChild(matches[i][j]);
                    grid[matches[i][j].col][matches[i][j].row] = null;
                    affectAbove(matches[i][j]);
               }
          }
     }
```

The `findAndRemoveMatches` function has two more tasks to perform. First, it calls `addNewPieces` to replace any missing `Pieces` in a column. Then, it calls `lookForPossibles` to make sure there are still more moves remaining. It only needs to do this if no matches were found. This would only happen if `findAndRemoveMatches` was called after new `Pieces` finished dropping and no current matches were found.

```
        // add any new piece to top of board
        addNewPieces();

        // no matches found, maybe the game is over?
        if (matches.length == 0) {
            if (!lookForPossibles()) {
                endGame();
            }
        }
    }
```

The `lookForMatches` Function

The `lookForMatches` function still has a pretty formidable task to perform. It must create an array of all the matches found. It must look for both horizontal and vertical matches of more than two `Pieces`. It does this by looping through the rows first, and then the columns. It only needs to check the first six spaces in each row and column, because a match starting in the seventh space can only be two in length, and the eighth space doesn't have anything following it at all.

The `getMatchHoriz` and `getMatchVert` functions take the delegated task of determining how long a match is starting at a location in the grid. For instance, if spot 3,6 is `Piece` type 4, and 4,6 is also type 4, but 5,6 is type 1, calling `getMatchHoriz(3,6)` should return 2, because the spot 3,6 starts a run of 2 matching `Piece` types.

If a run is found, we also want to push the loop forward a few steps. So, if there is a four-in-a-row match at 2,1, 2,2, 2,3, and 2,4, we just check 2,1 and get a result of 4, and then skip 2,2 2,3 and 2,4 to start looking again at 2.5.

Every time a match is found by `getMatchHoriz` or `getMatchVert`, they return an array containing each `Piece` in the match. These arrays are then added to the `matches` array in `lookForMatches`, which is in turn returned to whatever called `lookForMatches`:

```
//return an array of all matches found
public function lookForMatches():Array {
    var matchList:Array = new Array();

    // search for horizontal matches
    for (var row:int=0;row<8;row++) {
        for(var col:int=0;col<6;col++) {
            var match:Array = getMatchHoriz(col,row);
            if (match.length > 2) {
                matchList.push(match);
                col += match.length-1;
            }
        }
    }
    // search for vertical matches
    for(col=0;col<8;col++) {
        for (row=0;row<6;row++) {
            match = getMatchVert(col,row);
            if (match.length > 2) {
                matchList.push(match);
                row += match.length-1;
            }

        }
    }
    return matchList;
}
```

The `getMatchHoriz` and `getMatchVert` Functions

The `getMatchHoriz` function now has a specialized step to perform. Given a column and a row, it checks the next `Piece` over to see whether the `Piece` types match. If it does, it gets added to an array. It keeps moving horizontally until it finds one that doesn't match. Then, it returns the array it compiled. This array may only end up holding one `Piece`, the one at the original column and row, if the next one over doesn't match. But, for example, if it does match, and the next one does, too, it returns a run of three `Pieces`:

```
// look for horizontal matches starting at this point
public function getMatchHoriz(col,row):Array {
    var match:Array = new Array(grid[col][row]);
    for(var i:int=1;col+i<8;i++) {
        if (grid[col][row].type == grid[col+i][row].type) {
            match.push(grid[col+i][row]);
        } else {
            return match;
        }
    }
    return match;
}
```

The `getMatchVert` function is almost identical to the `getMatchHoriz` function, except that it searches along columns rather than rows:

```
// look for vertical matches starting at this point
public function getMatchVert(col,row):Array {
    var match:Array = new Array(grid[col][row]);
    for(var i:int=1;row+i<8;i++) {
        if (grid[col][row].type == grid[col][row+i].type) {
            match.push(grid[col][row+i]);
        } else {
            return match;
            }
        }
        return match;
    }
```

The affectAbove Function

We'll continue to work to build findAndRemoveMatches all the functions it needs. Next on the list is affectAbove. We pass a Piece into this, and then expect it to tell all Pieces above it to move down on step. In effect, it is a Piece saying, "I'm going away now, so all you guys drop down to fill in the gap."

A loop looks at the Pieces in the column that are above the current one. So, if the current one is 5,6, it looks at 5,5, 5,4, 5,3, 5,2, 5,1, and 5,0 in that order. The row of these Pieces will be incremented by one. Also, the Piece will tell the grid that it is in a new location.

Remember that with movePieces, we don't need to worry about how a Piece will animate to get to a new location, we just change the col or row and it will take care of itself:

```
// tell all pieces above this one to move down
public function affectAbove(piece:Piece) {
    for(var row:int=piece.row-1;row>=0;row--) {
        if (grid[piece.col][row] != null) {
            grid[piece.col][row].row++;
            grid[piece.col][row+1] = grid[piece.col][row];
            grid[piece.col][row] = null;
        }
    }
}
```

The addNewPieces Function

The next function we need to build is addNewPieces. This looks at each column, and then at each spot in the grid for each column, and counts the number of spots set to null. For each one, a new Piece is added. Although its col and row value is set to match its final destination, the y value is set to be above the top row, so it appears to fall down from above. Also, the isDropping Boolean is turned to true to indicate animation in progress:

```
// if there are missing pieces in a column, add one to drop
public function addNewPieces() {
    for(var col:int=0;col<8;col++) {
        var missingPieces:int = 0;
        for(var row:int=7;row>=0;row--) {
            if (grid[col][row] == null) {
                    var newPiece:Piece = addPiece(col,row);
                    newPiece.y = offsetY-spacing-spacing*missingPieces++;
                    isDropping = true;
                }
            }
        }
    }
```

Finding Possible Moves

As tricky as finding matches is, it is easier than finding possible matches. These aren't three-in-a-row matches, but rather the *possible* three-in-a-row matches.

The simplest answer is to scan the entire board, making every swap: 0,0 with 1,0, then 1,0 with 2,0, and so on. With each swap, check for matches. As soon as a swap that leads to a valid match is made, we can stop looking and return `true`.

This brute-force approach would work, but it could be awfully slow, especially on older machines. There is a better way.

If you think about what it takes to make a match, some patterns form. Typically, you have two `Pieces` of the same type in a row. The spot next to these `Pieces` is of a different type, but can be swapped in three directions to bring in another `Piece` that may match. Alternatively, you could have two `Pieces` spaced one apart from each other, and a swap could bring a matching `Piece` between them.

Figure 8.9 shows these two patterns, broken further into six possible patterns. Horizontally, the missing `Piece` in the match can come at the left or right, whereas vertically, it can come at the top or bottom.

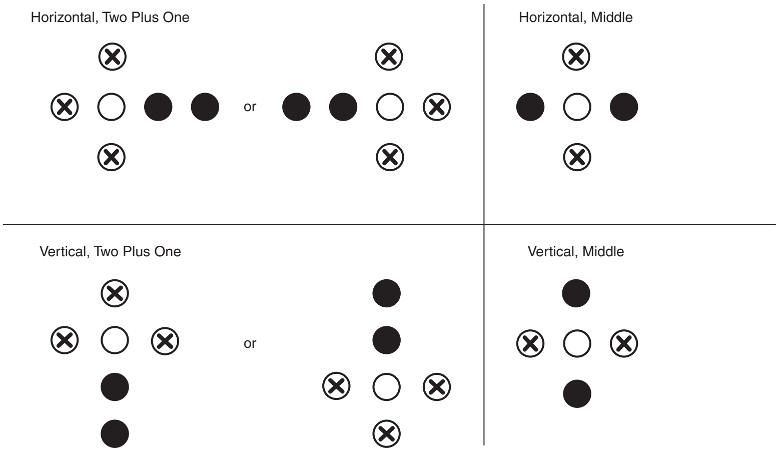


Figure 8.9
The filled circles represent the `Pieces` that will stay put. The empty circle represents the space that must be filled with the matching `Piece`. The circles with the X in them are possible locations for this matching `Piece`.

Knowing that there are only a few patterns that we need to look for, we can write a function that takes a list of locations and determines whether the pattern matches. With top-down programming, we can first write `lookForPossibles` and worry about writing the pattern-matching function later.

So, looking at the first pattern in Figure 8.9, we've got two spots that need to contain a match, and three spots where if any one of them contains the same as the matching type, we've got a positive result. Using the leftmost filled circle as point 0,0, we can say that the one next to it (1,0) must match. Then, there needs to be at least one other matching `Piece` at locations –1,–1, –2,0, or –1,1. Alternatively, the match can be on the right side of the initial pair. These would be positions 2,–1, 2,1, and 3,0.

So, there is a starting `Piece`. Then, a single position that must match the starting `Piece`. Then, six other positions where at least one must match. Figure 8.10 shows this as a diagram.

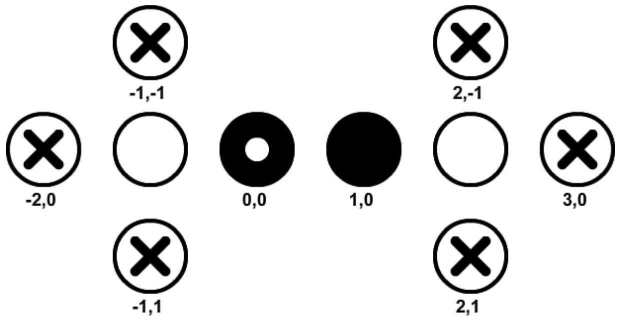


Figure 8.10
Position 1,0 needs to match 0,0. At least one of the six X spots also needs to match 0,0.

The function call would pass in an array of positions that must match, and a second array of positions where at least one must match. It would look like this:

```
matchPattern(col, row,  [[1,0]],  [[-2,0],[-1,-1],[-1,1],[2,-1],[2,1],[3,0]]))
```

We need a similar function call to deal with the "Horizontal, Middle" situation shown in Figure 8.9. Then, both the vertical patterns, too. The `lookForPossibles` searches for all of them, at all positions in the `grid`:

```
// look to see whether a possible move is on the board
public function lookForPossibles() {
     for(var col:int=0;col<8;col++) {
          for(var  row:int=0;row<8;row++) {

               // horizontal possible, two plus one
               if (matchPattern(col, row,
                       [[1,0]], [[-2,0],[-1,-1],[-1,1],[2,-1],[2,1],[3,0]])) {
                    return true;
               }

               // horizontal possible, middle
               if (matchPattern(col, row, [[2,0]], [[1,-1],[1,1]])) {
                return true;
            }

            // vertical possible, two plus one
            if (matchPattern(col, row,
                    [[0,1]], [[0,-2],[-1,-1],[1,-1],[-1,2],[1,2],[0,3]])) {
                return true;
            }

            // vertical possible, middle
            if (matchPattern(col, row, [[0,2]], [[-1,1],[1,1]])) {
                return true;
            }
        }
    }

    // no possible moves found
    return false;
}
```

The `matchPattern` function, although it has a large task to perform, is not a very large function. It needs to get the `type` of the `Piece` at the column and row position specified. Then, it looks through the `mustHave` list and checks the `Piece` in the relative position. If it doesn't match, there is no point continuing, and the function returns `false`.

Otherwise, each of the `Pieces` in `needOne` is checked. If any of them match, the function returns `true`. If none match, the function ends up returning `false`:

```
public function matchPattern(col,row:uint, mustHave, needOne:Array) {
    var thisType:int = grid[col][row].type;

    // make sure this has all must-haves
    for(var i:int=0;i<mustHave.length;i++) {
        if (!matchType(col+mustHave[i][0],
                row+mustHave[i][1], thisType)) {
            return false;
        }
    }

    // make sure it has at least one need-ones
    for(i=0;i<needOne.length;i++) {
        if (matchType(col+needOne[i][0],
                row+needOne[i][1], thisType)) {
            return true;
        }
    }
    return false;
}
```

All the comparisons in matchPattern are made through calls to matchType. The reason for this is that we are often trying to look at Pieces that are not in the grid. For instance, if the column and row passed into matchPattern are 5,0, and the Piece that is offset by –1,–1 is examined, we are looking up grid[4,-1], which is undefined, because there is no such thing as item –1 of an array.

The matchType function checks for grid location values outside of what we have set up, and returns a false instantly if that happens. Otherwise, the grid value is examined, and true is returned if the types match:

```
public function matchType(col,row,type:int) {
            // make sure col and row aren't beyond the limit
    if ((col < 0) || (col > 7) || (row < 0) || (row > 7)) return false;
    return (grid[col][row].type == type);
}
```

Score Keeping and Game Over

Way back in findAndRemoveMatches, we called addScore to award the player some points. This simple function adds points to the player's score, and relays the change to the text field on the screen:

```
public function addScore(numPoints:int) {
    gameScore += numPoints;
    MovieClip(root).scoreDisplay.text = String(gameScore);
}
```

When no possible matches are left, the endGame function takes the main timeline to the "gameover" screen. It also uses swapChildIndex to put the gameSprite at the very back, and so the sprites on the "gameover" frame will be above the game grid.

We need this because we won't be deleting the game grid at the end of the game. Instead, we'll leave it there for the player to examine:

```
public function endGame() {
    // move to back
    setChildIndex(gameSprite,0);
    // go to end game
    gotoAndStop("gameover");
}
```

We get rid of the grid and the gameSprite when the player is ready to move on. For that purpose, the cleanUp function takes care of it:

```
public function cleanUp() {
    grid = null;
    removeChild(gameSprite);
    gameSprite = null;
    removeEventListener(Event.ENTER_FRAME,movePieces);
}
```

In the main timeline, the function tied to the Play Again button calls `cleanUp` just before jumping back to the previous frame to start a new game.

Modifying the Game

One important decision to make is whether you want six or seven piece variations in the game. Most Match Three games seem to use six. I've used seven in the past, and that has worked, too. Using seven brings the game to an end sooner.

Bonus points are another improvement that can be made. An additional graphics layer can be added to the `Pieces`, similar to the selection border. It can be made visible on random `Pieces` to indicate bonus points. A `bonus` property can be added to the `Piece`, too, and it could trigger a second call to `addScore` when that `Piece` is removed.

Hints are a way to make the game more enjoyable for the player. When `lookForPossibles` is called, it calls `matchType` a number of times. If a possible match is found in the second loop inside `matchType`, a `true` is returned. The very position that `matchType` is examining at this point is a `Piece` that can be used in a swap to make a match. This can be placed in a new variable called something like `hintLocation`, and then that location used to highlight a `Piece` when the player clicks a hint button.

```
L P F F S F L H A E G M K W J
F Z L T I R T N B Q Y L O P O
H W F U G H A E S I R W S C G
K I Y F T K W M W I G R A I K
N U V U C O S U N E V U X M H
M E R C U R Y W B O R D G F F
E X P L T M W C J A J S M F X
S N W S Z C Y K N V O P L Q B
B A U X Z K Z U C T K R Q G H
F R I T H W S S P W X E Q I T
P J X K P Z D A N Z E T E V U
Y H X C N E W T J A G I L G Y
H T R A E J N U V K M P F C I
T U I M Z S S R Q A H U Q H U
B H N I D U O N T A X J G H Y
```

9

Word Games: Hangman and Word Search

- Strings and Text Fields
- Hangman
- Word Search

Using letters and words for games has been popular since the mid-20th century with board games, such as Scrabble, and paper games, such as crosswords and word searches.

These games work well as computer games and as web-based games. This chapter looks at two traditional games: hangman and word search. First, however, we need to take a closer look at how ActionScript handles strings and text fields.

Strings and Text Fields

SOURCE FILES
http://flashgameu.com
A3GPU09_TextExamples.zip

Before trying to make word games, it is worthwhile to see how ActionScript 3.0 handles strings and text fields. After all, we'll be using them quite a bit in the games.

ActionScript 3.0 String Handling

A `String` variable in ActionScript is a sequence of characters. We've been using strings throughout the book so far, without thinking much about how to perform advanced operations on them.

Creating a string is as simple as assigning some characters, surrounded by quotes, to a variable of type `String`:

```
var myString:String = "Why is a raven like a writing desk?";
```

String Deconstruction

We can deconstruct the string with a variety of functions. To get a single character at a location, we can use `charAt`:

```
myString.charAt(9)
```

This returns "r".

NOTE
ActionScript starts counting character positions in strings at character 0. So, the 0 character in the example is "W", and the ninth character is "r".

We can also use `substr` to get one or more characters from the string. The first parameter is the starting position, and the second parameter is the number of characters to return:

```
myString.substr(9,5)
```

This returns "raven".

The `substring` function is an alternative that takes the start and end position as parameters. Then, it returns the character from the start position to one less than the end position:

```
myString.substring(9,14)
```

This also returns “raven”.

The `slice` function acts like the `substring` function, except for a how the values of the second parameter are interpreted. In `substring`, if the second parameter is less than the first, the parameters are reversed. So, `myString.substring(9,14)` is the same as `myString.substring(14,9)`.

The `slice` function enables you to use negative values for the second parameter. It then counts backward from the end of the string. So, `myString.slice(9,-21)` will return “raven”.

Both `substring` and `slice` allow you to leave out the second parameter to get the remainder of the string:

```
myString.slice(9)
```

This returns “raven like a writing desk?”.

Comparing and Searching Strings

To compare two strings, you just need to use the == operator:

```
var testString = "raven";
trace(testString == "raven");
```

This returns `true`. However, it is case sensitive, so the following returns `false`:

```
trace(testString == "Raven");
```

If you want to compare two strings, regardless of case, it is just a matter of converting one or both strings to only lower- or only uppercase. You can do this with `toUpperCase` and `toLowerCase`:

```
testString.toLowerCase() == "Raven".toLowerCase()
```

To find a string inside another string, you can use `indexOf`:

```
myString.indexOf("raven")
```

This returns 9. We can also use `lastIndexOf` to find the last occurrence of a string inside another string:

```
myString.indexOf("a")
myString.lastIndexOf("a")
```

The first returns a 7, and the second returns a 20. These match the first and last positions of the letter *a* in the string “Why is a raven like a writing desk?”

NOTE

You can also give `indexOf` and `lastIndexOf` a second parameter. This number tells it where in the string to start looking, instead of starting at the very beginning or very end.

Most of the time when you are using `indexOf`, you are not looking for the position of the string, but whether the string is there at all. If it is, `indexOf` returns a number, 0 or greater. If not, it returns a −1. So, you can determine whether one string is found inside another like this:

```
(myString.indexOf("raven") != -1)
```

Another way to find a string inside another string is the `search` function:

```
myString.search("raven")
```

This returns 9.

The `search` function can take a string as a parameter, as previously mentioned, but it can also take something called a regular expression.

NOTE

A regular expression is a pattern used to find/replace strings inside of other strings. Regular expressions are used in many programming languages and tools.

The subject of regular expressions is deep. So deep, in fact, that several 1,000+ pages books exist that only cover regular expressions. There are also plenty of websites that go into detail. Check out http://flashgameu.com for a page of links on the subject.

```
myString.search(/raven/);
```

This example is the simplest type of regular expression and is identical to the previous use of `search`. Notice that the `/` character is used rather than quotes to surround the characters.

You can also give the regular expression some options after the last slash. The most useful here would be an `i` for case insensitivity:

```
myString.search(/Raven/i);
```

This example returns 9, even though there is a capital *R*.

You can also use wildcards in regular expressions. For instance, the period character represents any character:

```
myString.search(/r...n/)
```

This returns 9 because the word *raven* matches the pattern of *r* followed by any three characters, followed by *n:*

```
myString.search(/r.*n/)
```

This also returns a 9 because the pattern is *r* followed by any number of characters, followed by an *n*.

Building and Modifying Strings

You can append to a string by using a + operator. ActionScript will figure out that it is a `String`, not a number, and append rather than add. You can also use += to perform a simple append:

```
myString = "Why is a raven like";
myString += " a writing desk?";
```

To place something before an existing string, you use code like this:

```
myString = "a writing desk?";
myString = "Why is a raven like "+myString;
```

Whereas the `search` function searches and returns an index value, the `replace` function takes a regular expression and uses it to replace a portion of the string:

```
myString.replace("raven","door mouse")
```

The result would be "Why is a door mouse like a writing desk?"

You can also use a regular expression in the first parameter. This allows things to get very complex, such as moving text around inside a string rather than bringing in replacement text:

```
myString.replace(/(raven)(.*)(writing desk)/g,"$3$2$1")
```

This code example looks for *raven* and *writing desk* within the string, separated by any number of characters. It then reorders the string, with the *writing desk* coming first, the *raven* coming last, and the same characters in between.

Converting between Strings and Arrays

Both strings and arrays are useful for storing lists of information. It is useful, therefore, to be able to convert between them.

For instance, if you have a string "apple,orange,banana", you might want to create an array from it. To do this, you can use the `split` command:

```
var myList:String = "apple,orange,banana";
var myArray:Array = myList.split(",");
```

You can reverse the process by using the `join` command:

```
var myList:String = myArray.join(",");
```

In both cases, the character passed into the function represents the character used to divide the items in the string. If you use the `join` command, the resulting string is patched together with commas between the items.

Summary of String Functions

Table 9.1 contains all of the `String` functions we have discussed, plus a few more.

Table 9.1 `String` Functions

Function	Syntax	Description
`charAt`	`myString.charAt(pos)`	Returns the character at the location
`charCodeAt`	`String.charCodeAt(pos)`	Returns the character code of the character at the location
`concat`	`myString.concat(otherString)`	Returns a new string with the second string appended to the first
`fromCharCode`	`String.fromCharCode(num)`	Returns the character from the character code
`indexOf`	`myString.indexOf (innerString,startPos)`	Returns the location of the inner string in the main string
`join`	`myArray.join(char)`	Combines the elements in an array to make a string
`lastIndexOf`	`myString.lastIndexOf (innerString,startPos)`	Returns the last location of the inner string in the main string
`match`	`myString.match(regexp)`	Returns the substring matching the pattern
`replace`	`myString.replace (regexp,replacement)`	Replaces the pattern
`search`	`myString.search(regexp)`	Finds the location of the substring matching the pattern
`slice`	`myString.slice(start,end)`	Returns the substring
`split`	`myString.split(char)`	Splits the string into an array
`string`	`String(notAString)`	Converts a number or other value to a string
`substr`	`myString.substr(start,len)`	Returns the substring
`substring`	`myString.substr(start,end)`	Returns the substring
`toLowerCase`	`myString.toLowerCase()`	Returns the string with lowercase letters
`toUpperCase`	`myString.toUpperCase()`	Returns the string with uppercase letters

Applying Text Formatting to Text Fields

To place text on the screen, you need to create a new `TextField`. We've used these fields in previous chapters to create text messages and score displays.

If you want to use anything but the defalt font and style, you also need to create a `TextFormat` object and assign it to the text field. And for the advanced use of text in games, we also need to look at including fonts in our movies.

The `TextFormat` Object

Creating a `TextFormat` object is usually done just before creating a `TextField`. Or, it could be done at the start of a class if you know you'll be using that format for several of the text fields you'll be creating.

All `TextFormat` really is, is a holder for a set of properties. These properties control the way text looks.

NOTE

In ActionScript, you can also create style sheets, similar to CSS used in HTML documents. But these are only useful for HTML-formated text fields. We'll only be using plain text fields in our games.

You have two choices when creating a `TextFormat`. The first is to simply create a blank `TextFormat` object, and then set each of the properties in it. The other choice is to define many of the most common properties in the `TextFormat` declaration.

Here is an example of the quick way of creating a `TextFormat`:

```
var letterFormat:TextFormat = new
     TextFormat("Monaco",36,0x000000,true,false,false,null,null,"center");
```

It is, of course, important to remember the exact order of parameters for `TextFormat`. It goes like this: `font`, `size`, `color`, `bold`, `italic`, `underline`, `url`, `target`, and `align`. You can include as few or as many of these as you want, as long as they are in order. Use `null` to skip any properties you don't want to set.

NOTE

In fact, the list of parameters is more extensive, but I have left them out of the preceding example: `leftMargin`, `rightMargin`, `indent`, and `leading`.

Here is the longer way of doing things:

```
var letterFormat:TextFormat = new TextFormat();
letterFormat.font = "Monaco";
letterFormat.size = 36;
letterFormat.color = 0x000000;
letterFormat.bold = true;
letterFormat.align = "center";
```

Notice that I left out the `italic` and `underline` properties, because `false` is the default value for both.

Table 9.2 summarizes all the `TextFormat` properties.

Table 9.2 `TextFormat` Properties

Property	Example Values	Description
`align`	`TextFormatAlign.LEFT,` `TextFormatAlign.RIGHT,` `TextFormatAlign.CENTER,` `TextFormatALign.JUSTIFY`	Text alignment
`blockIndent`	Number	Indentation of all lines of a paragraph
`bold`	`true/false`	Makes the text bold
`bullet`	`true/false`	Displays text as a bulleted list
`color`	Color	Color of the text (for example, 0x000000)
`font`	Font name	Which font to use
`indent`	Number	Indent of the first line of the paragraph only
`italic`	`true/false`	Makes the text italic
`kerning`	`true/false`	Turns on special character spacing in some fonts
`leading`	Number	Vertical spacing between lines
`leftMargin`	Number	Extra space to the left
`letterSpacing`	Number	Extra space between characters
`rightMargin`	Number	Extra space to the right
`size`	Number	Font size
`tabStops`	Array of numbers	Sets tab locations
`target`	String	The browser target of a link (for example, "_blank")
`underline`	`true/false`	Makes the text underlined
`url`	String	The URL of the link

Creating `TextField` Objects

After you have a format, you need a text field to apply it to. Creating a `TextField` is like creating a `Sprite`. In fact, they are both types of display objects. They can both be added to other `Sprites` and movie clips with `addChild`:

```
var myTextField:TextField = new TextField();
addChild(myTextField);
```

To assign a format to a field, the best way is to use the `defaultTextFormat` property:

```
myTextField.defaultTextFormat = letterFormat;
```

The alternative is to use the function `setTextFormat`. The problem with this is that when you set the `text` property of the field, the text formatting reverts to the default for that field:

```
myTextField.setTextFormat(letterFormat);
```

The advantage of `setTextFormat` is that you can add second and third parameters to specify the start and end characters for the formatting. You can format a piece of the text rather than the whole thing.

In games, we commonly use small text fields for things such as score, level, time, lives, and so on. These fields don't need multiple text formats, and they are updated often. So, setting the `defaultTextFormat` is the best way to go in most cases.

Beside `defaultTextFormat`, the next most important property for us is `selectable`. Most of the text fields we'll be using for games are for display purposes only, or are not meant to be clickable. We want to turn off `selectable` so that the cursor doesn't change when over the field and the user can't select the text.

NOTE

The `border` property of a text field is a useful way to check the size and location of a text field you create with ActionScript. For instance, if you only place one word or letter in a field, you won't be able to see how big the field really is without setting the `border` to `true`, at least temporarily while testing.

Table 9.3 points out some useful `TextField` properties.

Table 9.3 `TextField` Properties

Property	Values	Description
`autoSize`	`TextFieldAutoSize.LEFT`, `TextFieldAutoSize.RIGHT`, `TextFieldAutoSize.CENTER`, `TextFieldAutoSize.NONE`	Resizes the text field to fit the text you place in it

Table 9.3 Continued

Property	Values	Description
`background`	`true/false`	Whether there is a background fill
`backgroundColor`	Color	Color of the background fill (for example, 0x000000)
`border`	`true/false`	Whether there is a border
`borderColor`	Color	Color of the border (for example, 0x000000)
`defaultTextFormat`	`TextFormat` object	Defines the default text format used when new text is applied
`embedFonts`	`true/false`	Must be set to `true` to use embedded fonts
`multiline`	`true/false`	Must be set to `true` to contain multiple lines of text
`selectable`	`true/false`	If `true`, the user can select the text in the field
`text`	String	Sets the entire text contents of the field
`textColor`	Color	Sets the color of the text (for example, 0x000000)
`type`	`TextFieldType.DYNAMIC`, `TextFieldType.INPUT`	Defines whether the user can edit the text
`wordWrap`	`true/false`	Whether the text wraps

Fonts

If you are making a quick game as an example, or to show your friends, or just to prove a basic concept, you can stick with basic fonts. Most of the games in this book do that just to keep them simple.

If you are developing something for a client, however, or for your website, you should really import the fonts you are using into the library. Doing so makes your game independent of the fonts the users have on their machine. It will also allow you to use more advanced effects with fonts, such as rotation and alpha.

To import a font, go to the library and choose New Font from the Library drop-down menu. (We've done this before, in Chapter 7, "Direction and Movement: Space Rocks.")

After importing the font, and naming it, right-click in Windows and Ctrl-click on Mac to select the Linkage properties and have the font included in the movie.

NOTE

Forgetting to set the Linkage properties for a font is a common mistake. Unlike with movie clips, the Linkage properties does not appear in the standard properties dialog for fonts. So, it is hard to find and therefore easy to forget. No errors or warnings are given when trying to use fonts that you have forgotten to link.

Even after you embed some fonts, your text fields will not use them until you set the `embedFonts` property to `true`.

Now by using the fonts that are in your library, you can manipulate and animate text in various ways.

Animated Text Example

The files **`TextFly.fla`** and **`TextFly.as`** show a use of strings, text format, and text fields to create an animation. Nothing is in the movie file except the font. The stage is empty.

The **TextFly.as class** takes a string and breaks it into characters, producing a single `TextField` and `Sprite` for each character. It then animates these `Sprites`.

The class begins by defining a bunch of constants that will determine how the animation will perform:

```
package {
     import flash.display.*;
     import flash.text.*;
     import flash.geom.Point;
     import flash.events.*;
     import flash.utils.Timer;

     public class TextFly extends MovieClip {
          // constants to define animation
          static const spacing:Number = 50;
          static const phrase:String = "FlashGameU";
          static const numSteps:int = 50;
          static const stepTime:int = 20;
          static const totalRotation:Number = 360;
          static const startScale:Number = 0.0;
          static const endScale:Number = 2.0;
          static const startLoc:Point = new Point(250,0);
          static const endLoc:Point = new Point(50,100);
          private var letterFormat:TextFormat =
                    new TextFormat("Monaco",36,0x000000,true,false,
                    false,null,null,TextFormatAlign.CENTER);
```

It then goes on to define some variables to hold the `Sprites`, and the state of the animation:

```
        // variables to keep track of animation
        private var letters:Array = new Array();
        private var flySprite:Sprite;
        private var animTimer:Timer;
```

The construction function creates all the `TextField` and `Sprite` objects. It also starts off the animation by creating a `Timer`:

```
public function TextFly() {
    // one sprite to hold everything
    flySprite = new Sprite();
    addChild(flySprite);

    // create all the of the letters as text fields inside sprites
    for(var i:int=0;i<phrase.length;i++) {
        var letter:TextField = new TextField();
        letter.defaultTextFormat = letterFormat;
        letter.embedFonts = true;
        letter.autoSize = TextFieldAutoSize.CENTER;
        letter.text = phrase.substr(i,1);
        letter.x = -letter.width/2;
        letter.y = -letter.height/2;
        var newSprite:Sprite = new Sprite();
        newSprite.addChild(letter);
        newSprite.x = startLoc.x;
        newSprite.y = startLoc.y;
        flySprite.addChild(newSprite);
        letters.push(newSprite);
    }

    // start animating
    animTimer = new Timer(stepTime,numSteps);
    animTimer.addEventListener(TimerEvent.TIMER,animate);
    animTimer.start();
}
```

Then, with each step of the animation, the rotation and scale of the `Sprites` will be set:

```
public function animate(event:TimerEvent) {
    // how far along is the animation
    var percentDone:Number = event.target.currentCount/event.target.repeatCount;

    // change position, scale and rotation
    for(var i:int=0;i<letters.length;i++) {
        letters[i].x = startLoc.x*(1.0-percentDone) +
                 (endLoc.x+spacing*i)*percentDone;
```

```
            letters[i].y = startLoc.y*(1.0-percentDone) + endLoc.y*percentDone;
            var scale:Number = startScale*(1-percentDone)+endScale*percentDone;
            letters[i].scaleX = scale;
            letters[i].scaleY = scale;
            letters[i].rotation = totalRotation*(percentDone-1);
        }
    }
```

Figure 9.1 shows this animation in mid-action.

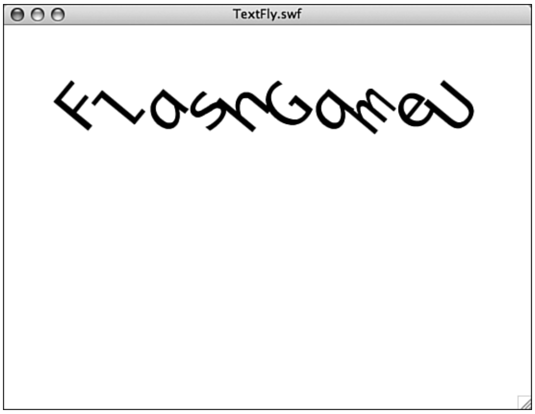


Figure 9.1
The TextFly program animates characters in text to have them fly in.

The ability to control text fields and formats at this level is important if you plan on making any games that use letters or words as playing pieces. Next, we take a look at hangman, perhaps the simplest letter game you could create.

Hangman

SOURCE FILES
http://flashgameu.com
A3GPU09_Hangman.zip

Hangman is not only one of the simplest word games, it is also very simple to program.

In keeping with the spirit of simplicity, the follow example will be a no-frills version of hangman.

Setting Up the Hangman

Traditionally, the game of hangman is played with two people. The first person makes up a word or phrase, and the second makes letter guesses. The first person draws out the word or phrase, using a blank space (underline) in place of each letter.

When the guesser guesses a letter used in the phrase, the first person fills in all the blank spaces where the letter is supposed to be. If the guesser picks a letter that is not used at all, the first person draws a bit more of a hanging man in a picture. Typically, it takes seven incorrect answers to complete the hanging man, which means the guesser loses.

NOTE

Hangman originated in the 19th century, around the time that gallows were used to punish criminals. This unusual image is still used in the game today, although any seven-part sequence can be substituted.

In our game, we'll use a seven-part sequence that differs a little from a hangman. Figure 9.2 shows our mascot character hanging on to a branch. If you make seven incorrect guesses, he'll fall.

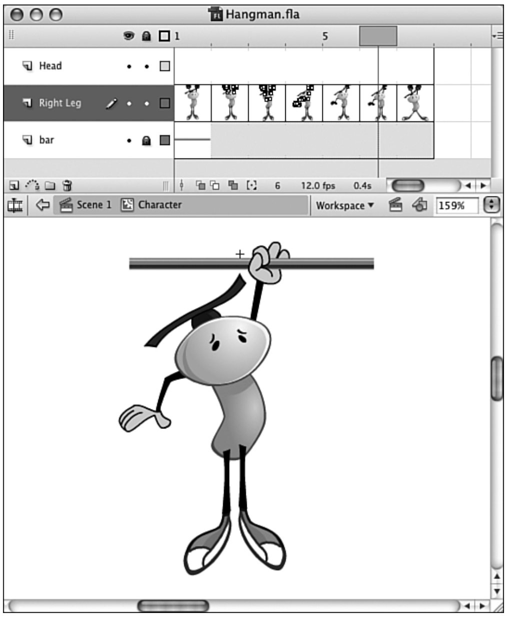


Figure 9.2
This seven-frame sequence can be substuted with just about any other similar idea.

So, the **`Hangman.fla`** movie has this one movie clip in it, and it is placed on the stage on the right side. Other than that, there isn't any other aspect to the movie except setting its class to `Hangman`.

The Hangman Class

This entire game is only about 50 lines of code. Only four class variables are needed. It is good to see that a fairly interesting game can be created so quickly and easily in ActionScript 3.0.

Two strings are needed: one to hold the phrase, and another to hold the display text, which will start with an underscore where the letters should be. Then we'll have one variable hold a reference to the text field, and another to keep track of the number of wrong guesses:

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

    public class Hangman extends Sprite {
        private var textDisplay:TextField;
        private var phrase:String =
                "Imagination is more important than knowledge.";
                            // - Albert Einstein
        private var shown:String;
        private var numWrong:int;
```

When the class starts, it creates a copy of the `phrase` by running it through the `replace` function with a regular expression. The expression `/[A-Za-z]/g` matches any letter character (*A* to *Z* and *a* to *z*, globally). It replaces these matches with an underscore:

```
public function Hangman() {
    // create a copy of text with _ for each letter
    shown = phrase.replace(/[A-Za-z]/g,"_");
    numWrong = 0;
```

The text field we'll set up will use a simple text format for Courier font, 30 point. It will set the `width` and `height` so that the text will not interfere with the hangman graphic to the right.

NOTE

The reason I chose Courier is that it is a monospaced font. This means that each letter has the same width. Other fonts have different widths for different letters (for example, *l* and *w*). By using a monospaced font, the text characters will not change positions as we substitute letters for the underscores.

```
    // set up the visible text field
    textDisplay = new TextField();
    textDisplay.defaultTextFormat = new TextFormat("Courier",30);
    textDisplay.width = 400;
    textDisplay.height = 200;
    textDisplay.wordWrap = true;
    textDisplay.selectable = false;
    textDisplay.text = shown;
    addChild(textDisplay);
```

The pressKey function will be assigned to the KEY_UP event for the stage:

```
     // listen for key presses
     stage.addEventListener(KeyboardEvent.KEY_UP,pressKey);
}
```

When the player presses a key, we'll use the event.charCode returned to get the letter pressed:

```
public function pressKey(event:KeyboardEvent) {
     // get letter pressed
     var charPressed:String = (String.fromCharCode(event.charCode));
```

After the letter is known, the phrase is searched for any matches. We're careful to use toLowerCase so that the key pressed will match both upper-and lowercase versions in the phrase.

When a match is found, the shown variable is updated by replacing the underscore in that position with the actual letter from phrase. This way, the uppercase letter is used if that is what is in phrase, and the lowercase letter if that is what is in phrase:

```
     // loop through and find matching letters
     var foundLetter:Boolean = false;
     for(var i:int=0;i<phrase.length;i++) {
          if (phrase.charAt(i).toLowerCase() == charPressed) {
               // match found, change shown phrase
               shown = shown.substr(0,i)+phrase.substr(i,1)+shown.substr(i+1);
               foundLetter = true;
          }
     }
```

The foundLetter Boolean is set to false when this search starts, and it is reset to true if any match is found. So, if it remains false, we know the letter wasn't in the phrase, and the hangman image will advance.

But first, we'll update the onscreen text by setting the text field to shown:

```
     // update on-screen text
     textDisplay.text = shown;

     // update hangman
     if (!foundLetter) {
          numWrong++;
          character.gotoAndStop(numWrong+1);
     }
}
```

NOTE

When testing in Flash, be sure to choose the menu option Control, Disable Keyboard Shortcuts. Otherwise, your key presses will not go through to the game window.

This short and simple game can be expanded to include the normal game elements we are used to: like a start and gameover screen. This quick game shows that you don't need to invest more than a few hours to create a fun game experience.

Now let's look at a more robust word game, the popular word search.

Word Search

SOURCE FILES
http://flashgameu.com
A3GPU09_WordSearch.zip

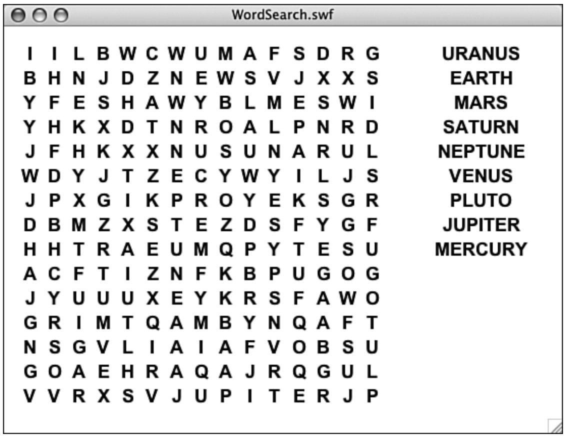
You would think that word searches have been around for a long time. In fact, they have only been here since the 1960s. They are popular on puzzle pages of newspapers, and sold in book collections.

Computer-based word search games can be generated randomly from a list of words or dictionaries. This makes them easier to create; you only need to come up with a list of words.

However, there are many challenging aspects to creating a computer word search game, such as displaying the letters; allowing for horizontal, vertical, and diagonal highlighting; and maintaining a word list.

Development Strategy

Our game will take a list of words and create a 15x15 grid of letters with those words hidden among other random letters. Figure 9.3 shows a complete grid.



```
I I L B W C W U M A F S D R G     URANUS
B H N J D Z N E W S V J X X S     EARTH
Y F E S H A W Y B L M E S W I     MARS
Y H K X D T N R O A L P N R D     SATURN
J F H K X X N U S U N A R U L     NEPTUNE
W D Y J T Z E C Y W Y I L J S     VENUS
J P X G I K P R O Y E K S G R     PLUTO
D B M Z X S T E Z D S F Y G F     JUPITER
H H T R A E U M Q P Y T E S U     MERCURY
A C F T I Z N F K B P U G O G
J Y U U U X E Y K R S F A W O
G R I M T Q A M B Y N Q A F T
N S G V L I A I A F V O B S U
G O A E H R A Q A J R Q G U L
V V R X S V J U P I T E R J P
```

Figure 9.3
The grid at the starting point, with the list of words to the right.

So we'll start with an empty grid and select random words from the list, random positions, and random directions. Then, we'll try to insert the word. If it doesn't fit, or it overlaps letters already placed into the grid, the placement is rejected and another random word, location, and direction are tried.

NOTE

Not all word search puzzles use all eight directions. Some do not have words backward, and others don't use diagonals. It is a matter of skill level. Simpler puzzles are good for young children, but are much too easy for adults.

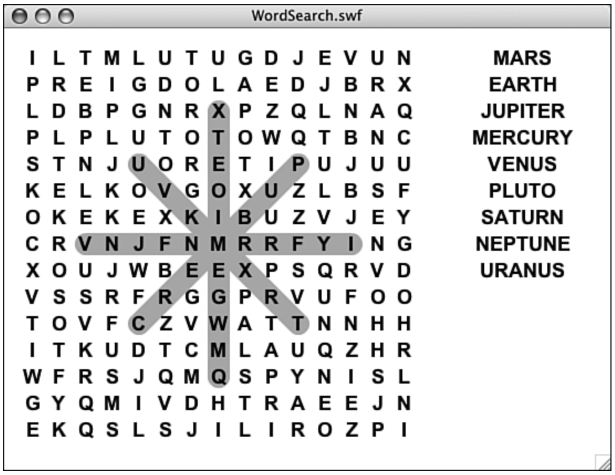
This loop repeats until either all the words are placed or a preset number of attempts have been performed. This will avoid cases where there is no more space left for a word. So, there is no guarantee that all the words will make it into the puzzle.

Our example uses only nine words, so it is unlikely to happen; but longer word lists will have trouble. Huge word lists will only use a sample of the words available each time, making the game more replayable by the same person.

After the words have been placed, all the unused letter positions are filled with random letters.

Also, a list of the words included are placed on the right side of the screen. As words are found, the ones in this list change color.

The player uses the mouse to click and drag on the grid. We'll be drawing a line under the letters to indicate which ones are selected. But, we'll only be doing this for valid selections. A valid selection would be horizontal, vertical, or at a 45-degree diagonal. Figure 9.4 demonstrates the different directions in which a word can be placed.



```
I L T M L U T U G D J E V U N      MARS
P R E I G D O L A E D J B R X      EARTH
L D B P G N R X P Z Q L N A Q      JUPITER
P L P L U T O T O W Q T B N C      MERCURY
S T N J U O R E T I P U J U U      VENUS
K E L K O V G O X U Z L B S F      PLUTO
O K E K E X K I B U Z V J E Y      SATURN
C R V N J F N M R R F Y I N G      NEPTUNE
X O U J W B E E X P S Q R V D      URANUS
V S S R F R G G P R V U F O O
T O V F C Z V W A T T N N H H
I T K U D T C M L A U Q Z H R
W F R S J Q M Q S P Y N I S L
G Y Q M I V D H T R A E E J N
E K Q S L S J I L I R O Z P I
```

Figure 9.4
Valid selections can go in eight different directions.

After all the words have been found, the game ends.

Defining the Class

The game frame in the movie is completely blank. Everything will be created with ActionScript. To do this, we need the `flash.display`, `flash.text`, `flash.geom` and `flash.events` class libraries:

```
package {
     import flash.display.*;
     import flash.text.*;
     import flash.geom.Point;
     import flash.events.*;
```

Several constants will make it easy to adjust the puzzle size, spacing between letters, outline line size, screen offset, and the text format:

```
     public class WordSearch extends MovieClip {
          // constants
          static const puzzleSize:uint = 15;
          static const spacing:Number = 24;
          static const outlineSize:Number = 20;
          static const offset:Point = new Point(15,15);
          static const letterFormat:TextFormat = new
                    TextFormat("Arial",18,0x000000,true,false,
                    false,null,null,TextFormatAlign.CENTER);
```

To keep track of the words and the grid of letters, we'll be using these three arrays:

```
          // words and grid
          private var wordList:Array;
          private var usedWords:Array;
          private var grid:Array;
```

The `dragMode` keeps track of whether the player is currently selecting a sequence of letters. The `startPoint` and `endPoint` will define that range of letters. The `numFound` will keep track of all the words found:

```
          // game state
          private var dragMode:String;
          private var startPoint,endPoint:Point;
          private var numFound:int;
```

This game will use several `Sprites`. The `gameSprite` holds everything. The others hold a particular type of element:

```
          // sprites
          private var gameSprite:Sprite;
          private var outlineSprite:Sprite;
          private var oldOutlineSprite:Sprite;
          private var letterSprites:Sprite;
          private var wordsSprite:Sprite;
```

Creating the Word Search Grid

The `startWordSearch` function has a lot of work to do in order to create a puzzle grid for use in the game. It will rely on the `placeLetters` function to do some of the work.

The `startWordSearch` Function

To start the game, we'll create an array with the words used in the puzzle. In this example, we'll use the nine planets, ignoring the International Astronomical Union's feelings about Pluto:

```
public function startWordSearch() {
     // word list
     wordList = ("Mercury,Venus,Earth,Mars,Jupiter,Saturn,Uranus,
          Neptune,Pluto").split(",");
```

Next, the `Sprites` are created. They are in the order in which they should be layered onto the stage. The outlines should be under the letters. Only the `gameSprite` is added to the stage; all the others are added to the `gameSprite`:

```
     // set up the sprites
     gameSprite = new Sprite();
     addChild(gameSprite);

     oldOutlineSprite = new Sprite();
     gameSprite.addChild(oldOutlineSprite);

     outlineSprite = new Sprite();
     gameSprite.addChild(outlineSprite);

     letterSprites = new Sprite();
     gameSprite.addChild(letterSprites);

     wordsSprite = new Sprite();
     gameSprite.addChild(wordsSprite);
```

The letter `Sprites` will be stored in the array `grid`. But, we'll first call `placeLetters` to get a nested array with the characters to be placed in these `Sprites`.

So, we are essentially dividing up the task of creating the game board into two steps. The first step will create a virtual grid of letters as a nested array. This will take care of adding the words from the word list and filling in the rest with random letters:

```
     // array of letters
     var letters:Array = placeLetters();
```

Now that we know where the letters will be placed, we need to create the `Sprites`, one for each letter. First, each letter gets a `TextField`. Then, this field is added to a new `Sprite`:

```
      // array of sprites
      grid = new Array();
      for(var x:int=0;x<puzzleSize;x++) {
           grid[x] = new Array();
           for(var y:int=0;y<puzzleSize;y++) {

                // create new letter field and sprite
                var newLetter:TextField = new TextField();
                newLetter.defaultTextFormat = letterFormat;
                newLetter.x = x*spacing + offset.x;
                newLetter.y = y*spacing + offset.y;
                newLetter.width = spacing;
                newLetter.height = spacing;
                newLetter.text = letters[x][y];
                newLetter.selectable = false;
                var newLetterSprite:Sprite = new Sprite();
                newLetterSprite.addChild(newLetter);
                letterSprites.addChild(newLetterSprite);
                grid[x][y] = newLetterSprite;
```

In addition to being created and added to `letterSprites`, each `Sprite` must get two events attached to it: `MOUSE_DOWN` and `MOUSE_OVER`. The first starts a selection, and the second allows the selection to be updated as the cursor moves over different letters:

```
                // add event listeners
                newLetterSprite.addEventListener(
                             MouseEvent.MOUSE_DOWN, clickLetter);
                newLetterSprite.addEventListener(
                             MouseEvent.MOUSE_OVER, overLetter);
           }
      }
```

When players release the mouse button, we can't be sure that they are over a letter at that moment. So, instead of attaching the `MOUSE_UP` event listener to the letters, we'll attach it to the `stage`:

```
   // stage listener
   stage.addEventListener(MouseEvent.MOUSE_UP, mouseRelease);
```

The last thing that needs to be created is the list of words to the right. This is just a collection of `TextField` objects placed in the `wordsSprite`. One is created for each word in the `usedWords` array. This array will be created by `placeLetters` and contain only the words that could fit into the puzzle:

```
      // create word list fields and sprites
      for(var i:int=0;i<usedWords.length;i++) {
           var newWord:TextField = new TextField();
           newWord.defaultTextFormat = letterFormat;
```

```
            newWord.x = 400;
            newWord.y = i*spacing+offset.y;
            newWord.width = 140;
            newWord.height = spacing;
            newWord.text = usedWords[i];
            newWord.selectable = false;
            wordsSprite.addChild(newWord);
        }
```

The game is ready to play, except for the `dragMode` and `numFound` variables that need to be set:

```
        // set game state
        dragMode = "none";
        numFound = 0;
    }
```

The `placeLetters` Function

The `placeLetters` function performs some challenging tasks. First, it creates an empty grid of 15x15 characters as a nested array. Each spot on the grid is filled with an `*`, which will signify an empty space in the puzzle:

```
    // place the words in a grid of letters
    public function placeLetters():Array {

        // create empty grid
        var letters:Array = new Array();
        for(var x:int=0;x<puzzleSize;x++) {
            letters[x] = new Array();
            for(var y:int=0;y<puzzleSize;y++) {
                letters[x][y] = "*";
            }
        }
```

The next step is to make a copy of the `wordList`. We want to use a copy, rather than the original, because we'll be removing words as we place them in the grid. We'll also be placing the words we use into a new array, `usedWords`:

```
        // make copy of word list
        var wordListCopy:Array = wordList.concat();
        usedWords = new Array();
```

Now it is time to add words into the grid. This is done by choosing a random word, random location, and a random direction. Then, an attempt will be made to place the word into the grid, letter by letter. If any conflict arises (for example, the edge of the grid is reached, or an existing letter in the grid doesn't match the letter we want to place there), the attempt is aborted.

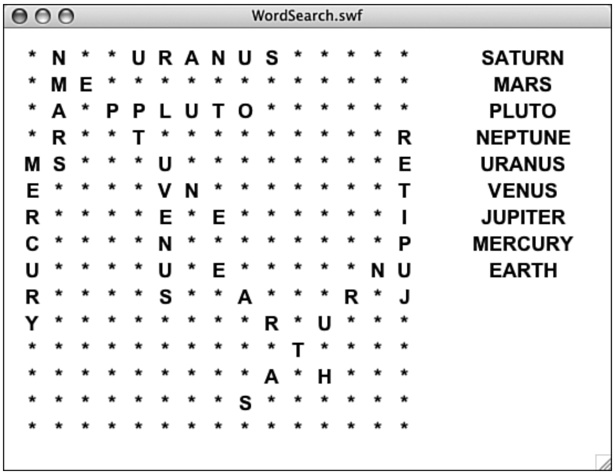
We'll keep trying, sometimes fitting a word in, and sometimes failing. We'll do this until the `wordListCopy` is empty. However, we'll also track the number of times we've tried in `repeatTimes`, which will start at 1,000 and decrease with every attempt. If `repeatTimes` reaches zero, we'll stop adding words. At that point, the chances are that every word that will fit into the puzzle is already there. We won't be using the rest of the words in this random build.

NOTE

We'll be using the technique of labeling the loops so that we can use the `continue` command to force the program to jump to the start of a loop outside of the current loop. Without these labels, it would be much harder to create the following code.

```
// make 1,000 attempts to add words
var repeatTimes:int = 1000;
repeatLoop:while (wordListCopy.length > 0) {
    if (repeatTimes-- <= 0) break;

    // pick a random word, location, and direction
    var wordNum:int = Math.floor(Math.random()*wordListCopy.length);
    var word:String = wordListCopy[wordNum].toUpperCase();
    x = Math.floor(Math.random()*puzzleSize);
    y = Math.floor(Math.random()*puzzleSize);
    var dx:int = Math.floor(Math.random()*3)-1;
    var dy:int = Math.floor(Math.random()*3)-1;
    if ((dx == 0) && (dy == 0)) continue repeatLoop;

    // check each spot in grid to see if word fits
    letterLoop:for (var j:int=0;j<word.length;j++) {
        if ((x+dx*j < 0) ¦¦ (y+dy*j < 0) ¦¦
                    (x+dx*j >= puzzleSize) ¦¦ (y+dy*j >= puzzleSize))
                     continue repeatLoop;
        var thisLetter:String = letters[x+dx*j][y+dy*j];
        if ((thisLetter != "*") && (thisLetter != word.charAt(j)))
                     continue repeatLoop;
    }

    // insert word into grid
    insertLoop:for (j=0;j<word.length;j++) {
        letters[x+dx*j][y+dy*j] = word.charAt(j);
    }

    // remove word from list
    wordListCopy.splice(wordNum,1);
    usedWords.push(word);
}
```

Now that we've got real words in the grid, the grid looks something like Figure 9.5, which is a game that leaves out this next step.



```
* N * * U R A N U S * * * * *      SATURN
* M E * * * * * * * * * * * *       MARS
* A * P P L U T O * * * * * *       PLUTO
* R * * T * * * * * * * * * R      NEPTUNE
M S * * * U * * * * * * * * E      URANUS
E * * * * V N * * * * * * * T       VENUS
R * * * * E * E * * * * * * I      JUPITER
C * * * * N * * * * * * * * P      MERCURY
U * * * * U * E * * * * * N U       EARTH
R * * * * S * * A * * * R * J
Y * * * * * * * * R * U * * *
* * * * * * * * * * T * * * *
* * * * * * * * * A * H * * *
* * * * * * * * S * * * * * *
* * * * * * * * * * * * * * *
```

Figure 9.5
*This grid has * characters where the random letters will be placed.*

The next loops look at every character in the grid and replaces the `*` with a random letter:

```
// fill rest of grid with random letters
for(x=0;x<puzzleSize;x++) {
     for(y=0;y<puzzleSize;y++) {
          if (letters[x][y] == "*") {
               letters[x][y] = String.fromCharCode(
                                   65+Math.floor(Math.random()*26));
          }
      }
}
```

When the `placeLetters` function is done, it returns its array so that the `Sprites` can be built from it:

```
  return letters;
}
```

User Interaction

We'll be using listeners to track three different mouse actions: click down, roll over a new `Sprite`, and release.

Mouse Click

When the player clicks down on a letter, the position on the grid is determined and placed into `startPoint`. Also, `dragMode` is set to `"drag"`.

The `findGridPoint` function returns a `Point` with the position of the letter in the grid. We'll build that function later:

```
// player clicks down on a letter to start
public function clickLetter(event:MouseEvent) {
     var letter:String = event.currentTarget.getChildAt(0).text;
     startPoint = findGridPoint(event.currentTarget);
     dragMode = "drag";
}
```

Cursor Drag

Every time the cursor passes over a letter on the screen, the following `overLetter` function is called. However, it first checks for `dragMode` to be equal to `"drag"`. So, the bulk of the function only happens after the player has clicked down on a letter.

The current point is stored in the `endPoint`. Now that we have both a `startPoint` and an `endPoint`, we can check the range to see whether it is valid. We'll assume it isn't, by clearing the `outlineSprite` graphic layer first. If it is a valid range, however, `drawOutline` sets the `outlineSprite` graphic layer with a new line.

So, basically, the outline is removed and redrawn each time the cursor changes letters:

```
// player dragging over letters
public function overLetter(event:MouseEvent) {
     if (dragMode == "drag") {
          endPoint = findGridPoint(event.currentTarget);

          // if valid range, show outline
          outlineSprite.graphics.clear();
          if (isValidRange(startPoint,endPoint)) {
               drawOutline(outlineSprite,startPoint,endPoint,0xFF0000);
          }
     }
}
```

Mouse Release

When the player releases the mouse over a letter, the `dragMode` is set to `"none"`, and the outline is cleared. Then, assuming the range is valid, two functions are called to deal with the selection.

The `getSelectedWord` function takes the range and returns the letters in it. Then, the `checkWord` function will see whether this word is in the list and take action if it is:

```
// mouse released
public function mouseRelease(event:MouseEvent) {
     if (dragMode == "drag") {
          dragMode = "none";
```

```
            outlineSprite.graphics.clear();

            // get word and check it
            if (isValidRange(startPoint,endPoint)) {
                    var word = getSelectedWord();
                checkWord(word);
            }
        }
    }
```

Utility Functions

The `findGridPoint` function takes a letter `Sprite` and figures out which location it is at. Because the `Sprites` are created from scratch, they cannot have dynamic variables attached to them. Therefore, we can't store each `Sprite`'s `x` and `y` value with it.

Instead, we'll just look through the `grid` and find the item in the `grid` that matches the `Sprite`:

```
// when a letter is clicked, find and return the x and y location
public function findGridPoint(letterSprite:Object):Point {

    // loop through all sprites and find this one
    for(var x:int=0;x<puzzleSize;x++) {
        for(var y:int=0;y<puzzleSize;y++) {
            if (grid[x][y] == letterSprite) {
                return new Point(x,y);
            }
        }
    }
    return null;
}
```

To determine whether two points in the puzzle make up a valid range, we perform three tests. If they are both on the same row or column, the range is valid. The third test looks at the `x` and `y` difference. If they are equal, regardless of being positive or negative, the selection is a 45-degree diagonal:

```
// determine if range is in the same row, column, or a 45-degree diagonal
public function isValidRange(p1,p2:Point):Boolean {
    if (p1.x == p2.x) return true;
    if (p1.y == p2.y) return true;
    if (Math.abs(p2.x-p1.x) == Math.abs(p2.y-p1.y)) return true;
    return false;
}
```

Drawing an outline behind the letters should be one of the more challenging aspects of this game. But sometimes you get lucky. Thanks to the rounded ends that are the default for lines, we can simply draw a line from one location to the other, make it nice and thick, and end up with a great-looking outline.

Note that some compensation is needed to place the ends of the line in the center of the letters. The locations of the letters corresponds to the upper left of the `TextField`, and thus the `Sprite` of the letters. So, half the `spacing` constant is added to compensate:

```
// draw a thick line from one location to another
public function drawOutline(s:Sprite,p1,p2:Point,c:Number) {
     var off:Point = new Point(offset.x+spacing/2, offset.y+spacing/2);
     s.graphics.lineStyle(outlineSize,c);
     s.graphics.moveTo(p1.x*spacing+off.x ,p1.y*spacing+off.y);
     s.graphics.lineTo(p2.x*spacing+off.x ,p2.y*spacing+off.y);
}
```

Dealing with Found Words

When a player finishes a selection, the first thing that happens is a word must be created from the letters in their selection. To do this, we'll determine the `dx` and `dy` between the two points, which helps us pick the letters from the grid.

Starting from the `startPoint`, we'll move one letter at a time. If the `dx` value is positive, each step means moving over one column to the right. If negative, it means a step to the left. Same for `dy` and up and down. This will take us in any of the eight possible directions of a valid selection.

The end result is a string of letters, the same letters seen in the selection on screen:

```
// find selected letters based on start and end points
public function getSelectedWord():String {

     // determine dx and dy of selection, and word length
     var dx = endPoint.x-startPoint.x;
     var dy = endPoint.y-startPoint.y;
     var wordLength:Number = Math.max(Math.abs(dx),Math.abs(dy))+1;

     // get each character of selection
     var word:String = "";
     for(var i:int=0;i<wordLength;i++) {
          var x = startPoint.x;
          if (dx < 0) x -= i;
          if (dx > 0) x += i;
          var y = startPoint.y;
          if (dy < 0) y -= i;
          if (dy > 0) y += i;
```

```
            word += grid[x][y].getChildAt(0).text;
        }
        return word;
    }
```

After we know the word the user thinks he has found, we can loop through the `usedWords` array and compare the found letters to the words. We must compare them both forward and backward. We don't want to place the restriction on the players that they must select the first letter first, especially because we'll be showing them some words reverse on the grid.

To reverse a word, a quick way to do it is to use `split` to convert the string to an array, then `reverse` to reverse the array, and then `join` to turn the array back into a string. Both `split` and `join` take `""`, a blank string, as the separator, because we want every character to be its own item in the array:

```
// check word against word list
public function checkWord(word:String) {

    // loop through words
    for(var i:int=0;i<usedWords.length;i++) {

        // compare word
        if (word == usedWords [i].toUpperCase()) {
            foundWord(word);
        }

        // compare word reversed
        var reverseWord:String = word.split("").reverse().join("");
        if (reverseWord == usedWords [i].toUpperCase()) {
            foundWord(reverseWord);
        }
    }
}
```

When a word is found, we want to permanently outline it and remove it from the list on the right.

The `drawOutline` function can draw the line on any `Sprite`. So, we'll have it draw the line this time to `oldOutlineSprite` (using a lighter shade of red).

Then, we'll loop through the `TextField` objects in `wordsSprite` and look at the `text` property of each. If this matches the word, the `TextField`'s color is changed to a light gray.

We'll also increase numFound and call endGame if all the words have been found:

```
// word found, remove from list, make outline permanent
public function foundWord(word:String) {

     // draw outline in permanent sprite
     drawOutline(oldOutlineSprite,startPoint,endPoint,0xFF9999);

     // find text field and set it to gray
     for(var i:int=0;i<wordsSprite.numChildren;i++) {
          if (TextField(wordsSprite.getChildAt(i)).text.toUpperCase() == word) {
               TextField(wordsSprite.getChildAt(i)).textColor = 0xCCCCCC;
          }
     }

     // see if all have been found
     numFound++;
     if (numFound == usedWords.length) {
          endGame();
     }
}
```

The endGame function simply takes the main timeline to "gameover". We don't want to erase the game Sprites yet, but rather have them appear under the "Game Over" message and Play Again button.

To make these items stand out better, I've placed them on a solid rectangle. Otherwise, they would just blend in with the grid of letters (see Figure 9.6).

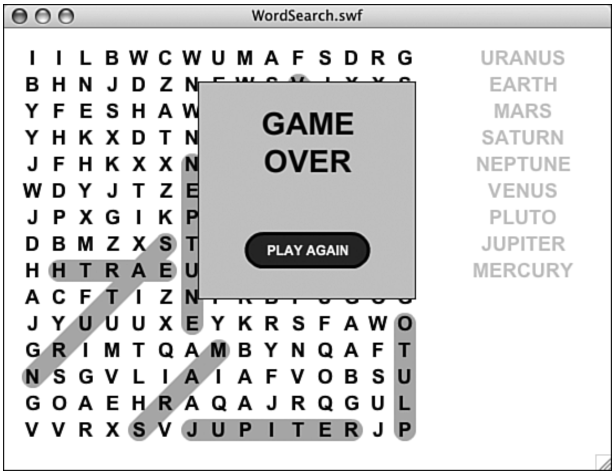


Figure 9.6
The rectangle helps the "Game Over" text and button stand out.

```
public function endGame() {
    gotoAndStop("gameover");
}
```

The Play Again button will call `cleanUp`, as well as go to the frame “play” to restart the game. Because we stored all our `Sprites` in the single `gameSprite Sprite`, we can just get rid of that and clear the `grid` to clean up:

```
public function cleanUp() {
    removeChild(gameSprite);
    gameSprite = null;
    grid = null;
}
```

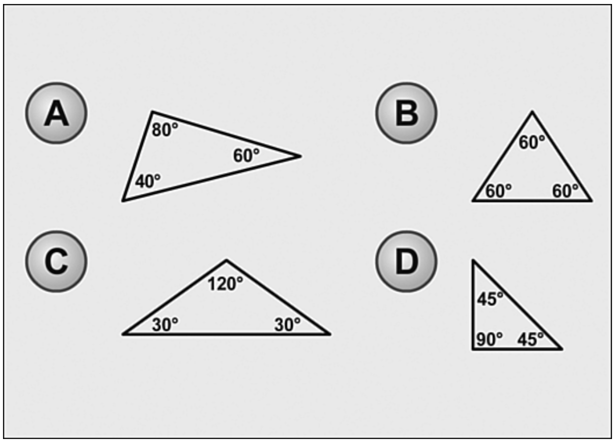
Modifying the Game

Players’ interest in this game may be strongly related to their interest in the words. You can create a puzzle for any subject. All it takes is a comma-separated word list.

In fact, you can use the technique from Chapter 2, “ActionScript Game Elements,” on including variables in the HTML code of a web page to pass in a short word list. Thus, a single word search game can be used on many pages of your site with a different word list.

You can also easily adjust the dimensions of the puzzle and the size and spacing of the letters. Doing so makes easier puzzles for children.

Another way to get word lists is to import them from external files. We’ll look at how to import external data in the next chapter.



10

Questions and Answers: Trivia and Quiz Games

- Storing and Retrieving Game Data
- Trivia Quiz
- Deluxe Trivia Quiz
- Picture Quiz

Different games can be used for different purposes. However, few games can be used for as diverse purposes as quiz games. You can have a quiz about almost any subject and at any difficulty level. The most difficult part about making quiz games is making them interesting. After all, a few multiple-choice questions is nothing more than a test. And few people like taking tests.

Quiz and trivia games are data driven. They rely on the questions and answers as primary game elements. This text data is best stored in external files and imported into the game dynamically. We'll look at strategies for doing this before starting on the games.

After that, we'll build a quiz game that takes an external text file and uses the questions and answers within for the game data. Then we'll go a step further and use external images in a picture quiz game.

Storing and Retrieving Game Data

SOURCE FILES
http://flashgameu.com
A3GPU10_XMLExamples.zip

A trivia game needs a list of questions and answers. The best way to bring in this data at the start of a game is by reading in an XML file.

NOTE
Flash prefers XML over other types of external data files. Flash can only read two types of files: XML and a list of variable assignments. The second is only really useful for small tasks. XML can be used for huge databases if need be.

Understanding XML Data

XML stands for eXtensible Markup Language. Its purpose is to have a simple format to be used to exchange information between systems.

If you've never seen an XML file before, but you have worked with HTML, you'll notice a similarity. Less than and greater than symbols are used in XML to enclose key defining words called *tags*. Take a look at this example:

```
<trivia>
     <item category="Entertainment">
          <question>Who is known as the original drummer of
                    the Beatles?</question>
          <answers>
               <answer>Pete Best</answer>
               <answer>Ringo Starr</answer>
               <answer>Stu Sutcliffe</answer>
```

```
            <answer>George Harrison</answer>
        </answers>
        <hint>Was fired before the Beatles hit it big.</hint>
        <fact>Pete stayed until shortly after their first
                audition for EMI in 1962, but was fired on
                August 16th of that year, to be replaced by
                Ringo Starr.</fact>
    </item>
</trivia>
```

This XML file represents a one-item trivia quiz. The data is in a nested format—tags inside of other tags. For instance, the entire document is one <*trivia*> object. Inside of that, is one `<item>`. In this `<item>` is one `<question>`, an `<answers>` object with four `<answer>` objects, and a `<hint>` and `<fact>` object.

NOTE

The individual objects in XML documents are also called *nodes*. A node can simply hold some data or it can have several *child* nodes. Some nodes have extra data associated with them, like the `item` node in the example has `category`. These are called *attributes*.

You can place an XML document right inside your ActionScript 3.0 code. For instance, the example movie **`xmlExample.fla`** has this in the frame 1 script:

```
var myXML:XML =
    <trivia>
        <item category="Entertainment">
            <question>Who is known as the original drummer of
                        the Beatles?</question>
            <answers>
                <answer>Pete Best</answer>
                <answer>Ringo Starr</answer>
                <answer>Stu Sutcliffe</answer>
                <answer>George Harrison</answer>
            </answers>
            <hint>Was fired before the Beatles hit it big.</hint>
            <fact>Pete stayed until shortly after their first
                        audition for EMI in 1962, but was fired on
                        August 16th of that year, to be replaced by
                        Ringo Starr.</fact>
        </item>
    </trivia>
```

Notice how no quotes or parenthesis were needed around the XML data. It can simply exist within ActionScript 3.0 code (although you can see how this might get unwieldy if the data were longer).

But now that we have some XML data in an `XML` object, we can play with how to extract information from it.

NOTE
XML data handling was vastly improved with ActionScript 3.0. Previously, you had to use more complex statements to find a specific node in the data. The new `XML` object in ActionScript 3.0 is different from the `XML` object in ActionScript 2.0, meaning that you can't directly convert from one to the other. So, beware of old code examples that might be in ActionScript 2.0 format.

To get the question node from the data, we would simply do this:

```
trace(myXML.item.question);
```

That's pretty straightforward. To get an attribute, you would use the `attribute` function:

```
trace(myXML.item.attribute("category"));
```

NOTE
A shortcut to getting the attribute is to use the `@` symbol. So, instead of `myXML.item.attribute("category")`, you can also write `myXML.item.@category`.

In the case of the `<answers>` node, we've got four answers. These can be treated like an array and accessed with brackets:

```
trace(myXML.item.answers.answer[1]);
```

Getting the number of nodes inside another node, like the `<answer>` nodes, is a little more obscure. But, it can be done like this:

```
trace(myXML.item.answers.child("*").length());
```

The `child` function returns a child of a node specified by a string or number. But using `"*"` returns all the child nodes. Then, using `length()` returns the number of child nodes. If you simply try to get the `length()` of a node, you'll only get 1 as a result because one node is always one node long.

Now that you know how to find your way around XML data, let's start dealing with larger XML documents imported from external files.

Importing External XML Files

When XML is saved as a file, it is similar to a plain-text file. In fact, you can open an XML file with most text editors. The file **trivia1.xml** is a short file with just ten trivia quiz items in it.

To open and read an external file, we'll use the `URLRequest` and `URLLoader` objects. Then, we'll set an event to trigger when the file has been loaded.

The following code sample shows XML loading code from **xmlimport.as**. The constructor function will create a `URLRequest` with the name of the XML file. Then, the `URLLoader` will start the download.

NOTE

You can pass any valid URL to `URLRequest`. Using just a filename, as we are here, means that the file should be next to the SWF Flash movie, in the same folder. However, you can specify a subfolder, or even use `../` and other path functions to give it a relative URL. You can also use absolute URLs. This works both on the server, and while testing locally on your machine.

We'll attach a listener to the `URLLoader`. This listener will call `xmlLoaded` when the file has been completely downloaded:

```
package {
    import flash.display.*;
    import flash.events.*;
    import flash.net.URLLoader;
    import flash.net.URLRequest;

    public class xmlimport extends MovieClip {
        private var xmldata:XML;

        public function xmlimport() {
            xmldata = new XML();
            var xmlURL:URLRequest = new URLRequest("xmltestdata.xml");
            var xmlLoader:URLLoader = new URLLoader(xmlURL);
            xmlLoader.addEventListener(Event.COMPLETE,xmlLoaded);
        }
```

The `xmlLoaded` function takes the data loaded from `event.target.data` and converts it to XML for storage in `xmldata`. As a test, it will put the second answer of the first question to the Output window:

```
        function xmlLoaded(event:Event) {
            xmldata  = XML(event.target.data);
            trace(xmldata.item.answers.answer[1]);
            trace("Data loaded.");
        }
    }
}
```

NOTE

XML objects are like arrays in that they are zero-based. So the first answer in the previous example is at position 0, and the second answer is at position 1.

Trapping Load Errors

Errors happen, and it is definitely useful to have some error checking. You can do this by adding another event to `URLLoader`:

```
xmlLoader.addEventListener(IOErrorEvent.IO_ERROR,xmlLoadError);
```

And then, you can get the error message from the `event` returned to `xmlLoadError`:

```
function xmlLoadError(event:IOErrorEvent) {
     trace(event.text);
}
```

However, I would not tell the end user the error message verbatim. For instance, if you just remove the file and try to run the movie, you get this error, followed by the filename:

```
Error #2032: Stream Error. URL: file:
```

Not an error message you want to show a player. Probably "Unable to load game file" is a better option.

Now you know how to retrieve larger XML documents, like the kind you will need to build trivia games.

Trivia Quiz

SOURCE FILES
http://flashgameu.com
A3GPU10_TriviaGame.zip

Trivia first became a form of entertainment in the 1950s with the advent of television. Quiz shows became popular and, if anything, have grown more popular over the years.

In the 1980s, board games like Trivial Pursuit became popular, allowing people to play trivia games, as well as watch them. Soon they became available on computers and the Internet.

Trivia games are a good way to address any subject in game form. Have a website about pirates? Make a pirate trivia game. Building a CD-ROM for a conference in Cleveland? Add a trivia game with interesting facts about the city.

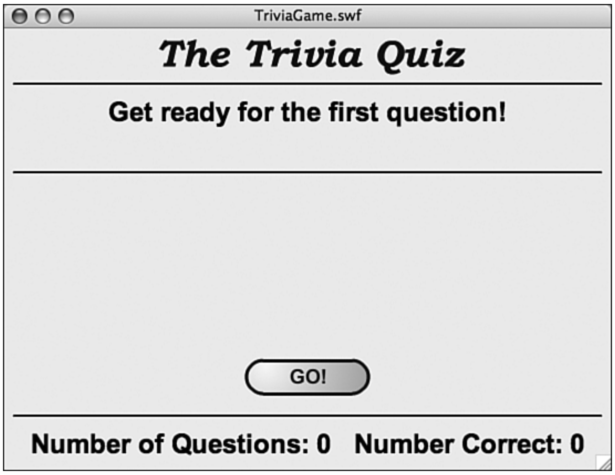
Let's build a simple quiz game first, and then go on to make a game with more bells and whistles later.

Designing a Simple Quiz Game

A basic trivia game is just a series of questions. The player reads one question, and then chooses an answer from several selections. Players get a point, or some sort of credit, if they get it right. Then, the game moves on to the next question.

We'll build this game like all of the rest: with three frames, the action taking placing in the middle frame.

The action, in this case, is a series of text and buttons. We'll start off by asking players if they are ready to go. They'll click a button to start (see Figure 10.1).



Get ready for the first question!

Number of Questions: 0 Number Correct: 0

Figure 10.1
At the start of the game, players are presented with a button they must click before the first question.

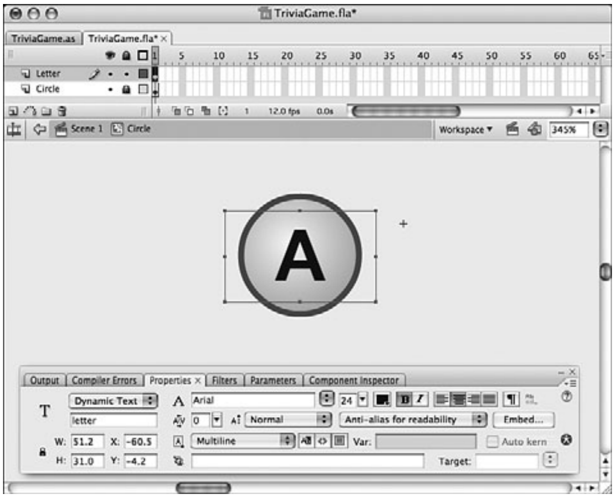
Next, they'll be presented with a question and four answers. The player must choose one answer. If the player gets it right, she will be told "You Got It!" If she is wrong, she will be told "Incorrect."

Either way, players get another button that they must press to advance to the next question.

Check out **TriviaGame.fla** and try playing to get a feel for how it goes. Now, let's build the game.

Setting Up the Movie

The movie file uses only two frames rather than the three we've been using. We'll need one new element in our movie library to make the quiz game. This will be a circle with a letter in it, which will display next to an answer. Figure 10.2 shows the movie clip.



Dynamic Text
letter
Arial 24
0 Normal
Anti-alias for readability
W: 51.2 X: -60.5
H: 31.0 Y: -4.2
Multiline
Var:
Target:

Figure 10.2
The `Circle` *movie clip contains a dynamic text field and a background circle.*

The text field in the `Circle` movie clip is named `letter`. We'll be creating four of these, one for each answer, and placing it next to the answer text. The `letter` in each will be different: *A*, *B*, *C*, or *D*.

NOTE

If you look closely at Figure 10.2, you can see the registration point for the movie clip off to the upper right. This will match the 0,0 location of the text field that will go next to it. This way, we can set the `Circle` and the answer text field to the same location, and they will appear next to each other rather than on top of one another.

The same technique of a background graphic and a text field will be used in the `GameButton` movie clip. This will allow us to use the same button movie clip for various buttons throughout the game.

The movie also contains some background graphics, notably a title and some horizontal lines (shown previously in Figure 10.1).

Setting Up the Class

Because this game loads the quiz data from an external file, we need some parts of the `flash.net` library to use the `URLLoader` and `URLRequest` functions:

```
package {
     import flash.display.*;
     import flash.text.*;
     import flash.events.*;
     import flash.net.URLLoader;
     import flash.net.URLRequest;
```

The game will use a variety of variables. We'll be putting the data loaded from the file into `dataXML`. We've also got several different text formats and some references to dynamic text fields that we'll be creating:

```
public class TriviaGame extends MovieClip {

    // question data
    private var dataXML:XML;

    // text formats
    private var questionFormat:TextFormat;
    private var answerFormat:TextFormat;
    private var scoreFormat:TextFormat;

    // text fields
    private var messageField:TextField;
    private var questionField:TextField;
    private var scoreField:TextField;
```

The plan for sprites is to have one `gameSprite` that contains everything. Inside of that, we'll have a `questionSprite` that holds all the elements of a single quiz question: a text field for the question and other sprites for the answers. The `answerSprites` will contain the text fields and `Circle` movie clips for each answer, which will be stored in their own sprites. We don't need a class variable to reference those, however, because they will be neatly stored in the `answerSprites` sprite.

There is also a reference for the `GameButton`, so that when we create a button, we can use this reference to remove it:

```
    // sprites and objects
    private var gameSprite:Sprite;
    private var questionSprite:Sprite;
    private var answerSprites:Sprite;
    private var gameButton:GameButton;
```

To keep track of game state, we need `questionNum`, which tracks the question we are on; `numCorrect`, which is essentially the player's score; and `numQuestionsAsked`, which is another aspect of the player's score.

To keep track of the question being asked, we'll put all four answers in random order into the `answers` array. Before we shuffle them, however, we'll take note of the original first answer, which should be the correct one, in the `correctAnswer` variable:

```
    // game state variables
    private var questionNum:int;
    private var correctAnswer:String;
    private var numQuestionsAsked:int;
    private var numCorrect:int;
    private var answers:Array;
```

The constructor function will create the gameSprite and then set all three TextFormat objects up:

```
 public function startTriviaGame() {

      // create game sprite
      gameSprite = new Sprite();
      addChild(gameSprite);

      // set text formats
      questionFormat = new TextFormat("Arial",24,0x330000,
           true,false,false,null,null,"center");
      answerFormat = new TextFormat("Arial",18,0x330000,
           true,false,false,null,null,"left");
      scoreFormat = new TextFormat("Arial",18,0x330000,
           true,false,false,null,null,"center");
```

NOTE

There is no way to duplicate a TextFormat object. If you simply set answerFormat = questionFormat and then make a change to one, it changes them both. So, it is important to make new TextFormat objects for each variable.

However, you can set a temporary variable, like myFont to a value like "Arial", and then use myFont in place of "Arial" in every TextFormat declaration. Then, you can alter the font used in the game with a single change in one place.

When the game starts, the scoreField and messageField are created. Instead of creating a TextField, adding it with addChild, and setting each of its properties for every piece of text we need, we'll make a utility function called createText that does this all for us in one line of code. For instance, the messageField will contain the text "Loading Questions …" using the format questionFormat. It places it in the gameSprite at 0,50 with a width of 550. We'll look at createText later on:

```
      // create score field and starting message text
      scoreField = createText("",questionFormat,gameSprite,0,360,550);
      messageField = createText("Loading Questions...",questionFormat,
           gameSprite,0,50,550);
```

After the game state is set, showScore is called to place the score text at the bottom of the screen. We'll look at that later, too.

Then xmlImport is called to retrieve the quiz data:

```
      // set up game state and load questions
      questionNum = 0;
      numQuestionsAsked = 0;
```

```
    numCorrect = 0;
    showScore();
    xmlImport();
}
```

The text “Loading Questions …” will appear on the screen and remain there until the XML document has been read. While testing, this might be less than a second. After the game is on a server, it should appear for a little longer, depending on the responsiveness of the player’s connection.

Loading the Quiz Data

Questions are loaded using functions similar to the example at the beginning of this chapter. No error checking is done, to keep things simple. The file **trivia1.xml** contains ten items:

```
// start loading of questions
public function xmlImport() {
    var xmlURL:URLRequest = new URLRequest("trivia1.xml");
    var xmlLoader:URLLoader = new URLLoader(xmlURL);
    xmlLoader.addEventListener(Event.COMPLETE, xmlLoaded);
}
```

After the loading is complete, the data is placed in `dataXML`. Then, the text message, which had been showing “Loading Questions …,” is removed. It is replaced with a new message that says “Get ready for the first question!”

Another utility function is called to create a `GameButton`. In this case, the button label “GO!” is placed inside the button. We’ll look at `showGameButton` a little later in this chapter:

```
// questions loaded
public function xmlLoaded(event:Event) {
    dataXML  = XML(event.target.data);
    gameSprite.removeChild(messageField);
    messageField = createText("Get ready for the first
         question!",questionFormat,gameSprite,0,60,550);
    showGameButton("GO!");
}
```

The game now waits for the player to click the button.

Message Text and Game Button

Several utility functions are needed in this game to create text fields and buttons. These cut down the amount of code needed quite a bit. We don’t have to repeat the same `new TextField`, `addChild`, and `x` and `y` settings every time we create a text field.

What `createText` does is take a series of parameters and make a new `TextField`. It sets the `x`, `y`, `width`, and `TextFormat` values to the values passed in as parameters. It also sets some constant parameters, such as `multiline` and `wordWrap`, which will be the same for everything created in the game.

The alignment of the text in the field will vary between centered and left justified. This is included in the `TextFormat`. However, we want to set the `autoSize` property of the field to the appropriate value, so a test is performed, and `autoSize` is set to either `TextFieldAutoSize.LEFT` or `TextFieldAutoSize.RIGHT`.

Finally, the `text` of the field is set, and the field is added to the sprite passed in as another parameter. The `TextField` is returned by the function, so we can set a variable to reference it for later removal:

```
// creates a text field
public function createText(text:String, tf:TextFormat,
          s:Sprite, x,y: Number, width:Number): TextField {
     var tField:TextField = new TextField();
     tField.x = x;
     tField.y = y;
     tField.width = width;
     tField.defaultTextFormat = tf;
     tField.selectable = false;
     tField.multiline = true;
     tField.wordWrap = true;
     if (tf.align == "left") {
          tField.autoSize = TextFieldAutoSize.LEFT;
     } else {
          tField.autoSize = TextFieldAutoSize.CENTER;
     }
     tField.text = text;
     s.addChild(tField);
     return tField;
}
```

One field that won't be created, destroyed, and then created again during the game is the `scoreField`. This field is created once and placed at the bottom of the screen. Then, we'll use `showScore` to update the `text` in the field:

```
// updates the score
public function showScore() {
     scoreField.text = "Number of Questions: "+numQuestionsAsked+
          "   Number Correct: "+numCorrect;
}
```

In the same way that `createText` enables us to create different types of text fields with one function, `showGameButton` allows us to create different buttons. It takes `buttonLabel` as a parameter and sets the `text` of the label inside the button to match. Then, it places the button on the screen.

The gameButton variable is already a class property, so it will be available for removeChild later on. We'll add an event listener to this button so that it calls pressGameButton when pressed. This will be used to advance the game:

```
// ask players if they are ready for next question
public function showGameButton(buttonLabel:String) {
     gameButton = new GameButton();
     gameButton.label.text = buttonLabel;
     gameButton.x = 220;
     gameButton.y = 300;
     gameSprite.addChild(gameButton);
     gameButton.addEventListener(MouseEvent.CLICK,pressedGameButton);
}
```

NOTE

With top-down programming, you want to test each bit of code as you write it. Unfortunately, the preceding code sample generates an error because pressedGameButton does not yet exist. At this point, I usually create a dummy pressedGameButton function that contains no code. That way I can test the placement of the button first, before needing to program what happens when the player clicks the button.

Moving the Game Forward

When the player clicks a button, the game should move forward one step. Most of the time, this means presenting the new question. However, if there are no more questions, the game ends.

First, we'll remove the previous question. If this is the first question, questionSprite has not yet been created. So, we'll check for the existence of questionSprite and only remove it if it is there:

```
// player is ready
public function pressedGameButton(event:MouseEvent) {
     // clean up question
     if (questionSprite != null) {
          gameSprite.removeChild(questionSprite);
     }
```

Other things must be removed, too. The message and button left over from the pause before or between questions is removed:

```
     // remove button and message
     gameSprite.removeChild(gameButton);
     gameSprite.removeChild(messageField);
```

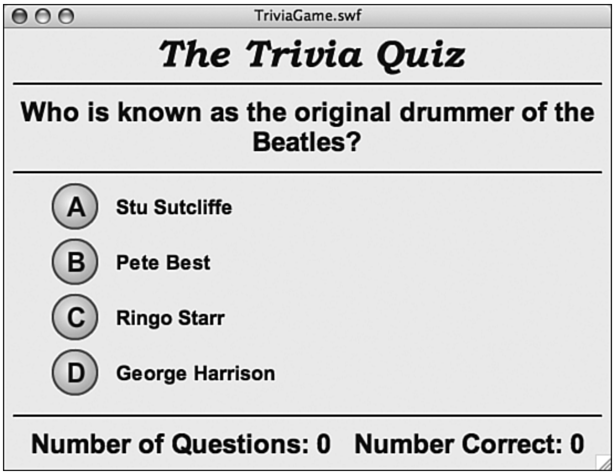
Now we must determine whether all the questions have been exhausted. If so, jump to the "gameover" frame at this point. The screen is already blank, from having the previous question, message, and button removed.

If this is not the end, call `askQuestion` to display the next question:

```
     // ask the next question
     if (questionNum >= dataXML.child("*").length()) {
         gotoAndStop("gameover");
     } else {
         askQuestion();
     }
 }
```

Displaying the Questions and Answers

The `askQuestion` function takes the next question from the quiz data and displays it. It puts everything it creates into the `questionSprite` sprite, which makes it easy to dispose of later on. Figure 10.3 shows the screen after a question has been displayed.



The Trivia Quiz

Who is known as the original drummer of the Beatles?

Stu Sutcliffe

Pete Best

Ringo Starr

George Harrison

Number of Questions: 0 Number Correct: 0

Figure 10.3
The question and four answers are displayed in the `questionSprite`*, which covers most of the middle of the screen.*

```
 // set up the question
 public function askQuestion() {
     // prepare new question sprite
     questionSprite = new Sprite();
     gameSprite.addChild(questionSprite);
```

The question itself will appear in a single field near the top of the screen:

```
     // create text field for question
     var question:String = dataXML.i       tem[questionNum].question;
     questionField = createText(question,questionFormat,questionSprite,0,60,550);
```

Before we place the answers, we need to shuffle them. The first answer in the original data is the correct one, so we'll store a copy of it in `correctAnswer`. Then, we'll call `shuffleAnswers` to get an array of all the answers, but in a random order:

```
     // create sprite for answers, get correct answer, and shuffle all
     correctAnswer = dataXML.item[questionNum].answers.answer[0];
     answers = shuffleAnswers(dataXML.item[questionNum].answers);
```

The answers are in a subsprite of `questionSprite` called `answerSprites`. Both a `TextField` and a `Circle` are created for each answer. The `Circle` objects are all assigned different letters, from *A* to *D*. They are both placed at the same location, but the `Circle` has been designed to appear to the left of its location, whereas the text will appear to the right.

Both the text and `Circle` will be bundled together in a single new sprite, and this sprite will get a `CLICK` listener assigned to it so that it can react like a button:

```
     // put each answer into a new sprite with a circle icon
     answerSprites = new Sprite();
     for(var i:int=0;i<answers.length;i++) {
          var answer:String = answers[i];
          var answerSprite:Sprite = new Sprite();
          var letter:String = String.fromCharCode(65+i); // A-D
          var answerField:TextField =
                   createText(answer,answerFormat,answerSprite,0,0,450);
          var circle:Circle = new Circle(); // from Library
          circle.letter.text = letter;
          answerSprite.x = 100;
          answerSprite.y = 150+i*50;
          answerSprite.addChild(circle);
          answerSprite.addEventListener(MouseEvent.CLICK,clickAnswer);
          answerSprite.buttonMode = true;
          answerSprites.addChild(answerSprite);
     }
     questionSprite.addChild(answerSprites);
}
```

NOTE

To convert from a number to a letter, `String.fromCharCode(65+i)` is used. It will get character 65 for *A*, character 66 for *B*, and so on.

The `shuffleAnswers` function takes an `XMLList`, which is the data type returned by asking for `dataXML.item[questionNum].answers`. It loops, removing one random item from the list at a time and placing it in an array. It then returns this randomly sorted array of answers:

```
// take all the answers and shuffle them into an array
public function shuffleAnswers(answers:XMLList) {
     var shuffledAnswers:Array = new Array();
     while (answers.child("*").length() > 0) {
            var r:int = Math.floor(Math.random()*answers.child("*").length());
            shuffledAnswers.push(answers.answer[r]);
            delete answers.answer[r];
        }
        return shuffledAnswers;
    }
```

Judging the Answers

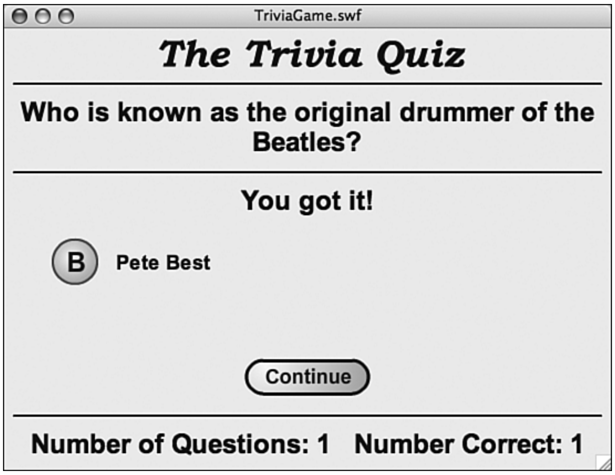
All the functions so far have just been setting up the game. Now, finally, the player is presented with the question, as shown previously in Figure 10.3.

When the player clicks any one of the four answers, `clickAnswer` is called. The first thing this function does is to get the text of the selected answer. The `TextField` is the first child of the `currentTarget`, so the value of the `text` property if grabbed and placed into `selectedAnswer`.

Then, this is compared with the `correctAnswer` that we stored when the question was displayed. If the player got it right, `numCorrect` is incremented. A new text message is displayed in either case:

```
    // player selects an answer
    public function clickAnswer(event:MouseEvent) {

        // get selected answer text, and compare
        var selectedAnswer = event.currentTarget.getChildAt(0).text;
        if (selectedAnswer == correctAnswer) {
            numCorrect++;
            messageField = createText("You got it!",
                    questionFormat,gameSprite,0,140,550);
        } else {
            messageField = createText("Incorrect! The correct answer was:",
                    questionFormat,gameSprite,0,140,550);
        }
        finishQuestion();
    }
```

Then all the answers are examined. The function `finishQuestion` loops through each sprite. The correct one is moved to a `y` position that places it in the middle. All event listeners are removed, too. The others are turned invisible. Figure 10.4 shows how the screen looks now.



Who is known as the original drummer of the Beatles?

You got it!

Pete Best

Number of Questions: 1 Number Correct: 1

Figure 10.4
The correct answer is moved to the middle and a message displayed.

```
public function finishQuestion() {
    // remove all but the correct answer
    for(var i:int=0;i<4;i++) {
        answerSprites.removeEventListener(MouseEvent.CLICK,clickAnswer);
        if (answers[i] != correctAnswer) {
            answerSprites.getChildAt(i).visible = false;
        } else {
            answerSprites.getChildAt(i).y = 200;
        }
    }
}
```

The score also needs to be updated, as well as the `questionNum` pointer. Finally, a new button is created with the label “Continue.” You can see it in Figure 10.4, too:

```
    // next question
    questionNum++;
    numQuestionsAsked++;
    showScore();
    showGameButton("Continue");
}
```

The button created by `clickAnswer` is the link back to the next question. When the player clicks it, `pressGameButton` is called, which triggers the next question, or the gameover screen.

Ending the Game

The “gameover” frame has a “Play Again” button that will jump the player back to the game. But first, it needs to call `cleanUp` to remove the remnants of the game:

```
// clean up sprites
public function cleanUp() {
    removeChild(gameSprite);
    gameSprite = null;
    questionSprite = null;
    answerSprites = null;
    dataXML = null;
}
```

Now the game is ready to be started all over again.

This simple quiz game is good enough for special interest websites or products that need something very basic. For a full-featured trivia game, however, we need to add a lot more.

Deluxe Trivia Quiz

SOURCE FILES
http://flashgameu.com
A3GPU10_TriviaGameDeluxe.zip

To improve upon what we already have, we'll add some features to make the game more exciting, challenging, and fun.

First, the player should have a time limit for answering questions. Most game shows and quizzes do this.

Second, we'll add a hint button to the quiz so that the player can get a little extra help. There are two types of hints, and we'll look at adding them both.

Next, we'll make the game more informative by placing a piece of extra information after every question. This will make the game more educational. The information will expand upon what the player just learned by answering the question.

Finally, we'll revamp the scoring system. This must take into account the time it takes to answer a question and whether the player requested a hint.

As an extra bonus, we'll make the quiz read in a large number of questions, but pick ten at random to use. This way the quiz differs each time it is played.

Adding a Time Limit

To add a time limit to the game, we need a visual representation of the time the player has to answer a question. We can make this a separate movie clip object. This `Clock` object can be any device for telling time: a clock face, some text, or something else.

For this example, I've set up a 26-frame movie clip. All frames contain 25 circles. Starting with frame 2, one of the circles is filled in with a solid shape. So on the first frame, all 25 circles are empty. On the 26th frame, all are filled. Figure 10.5 shows this `Clock` movie clip.

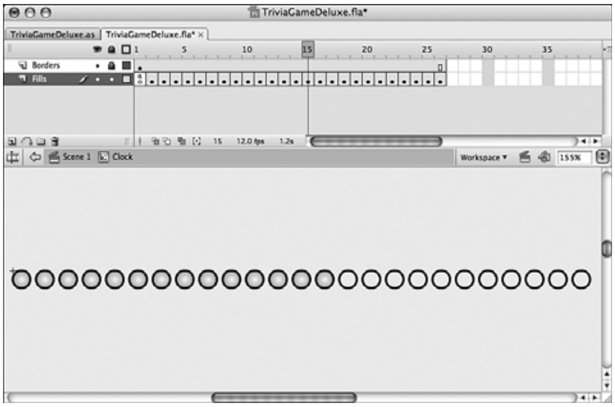


Figure 10.5
The 15th frame of the `Clock` movie clip shows 14 filled-in circles.

We'll use a `Timer` to count the seconds. We need to add that to the `import` statements:

```
import flash.utils.Timer;
```

Next, we add a `Clock` to the sprites being used:

```
private var clock:Clock;
```

And a `Timer`:

```
private var questionTimer:Timer;
```

In the `askQuestion function`, we need to add the `Clock` and start the `Timer`:

```
// set up a new clock
clock = new Clock();
clock.x = 27;
clock.y = 137.5;
questionSprite.addChild(clock);
questionTimer = new Timer(1000,25);
questionTimer.addEventListener(TimerEvent.TIMER,updateClock);
questionTimer.start();
```

The `Clock` will be positioned just under the question on the screen. In fact, we need to expand the height of the game a bit and move the elements down some to accommodate the `Clock` and some of the other elements we'll be adding soon. Figure 10.6 shows the new layout.

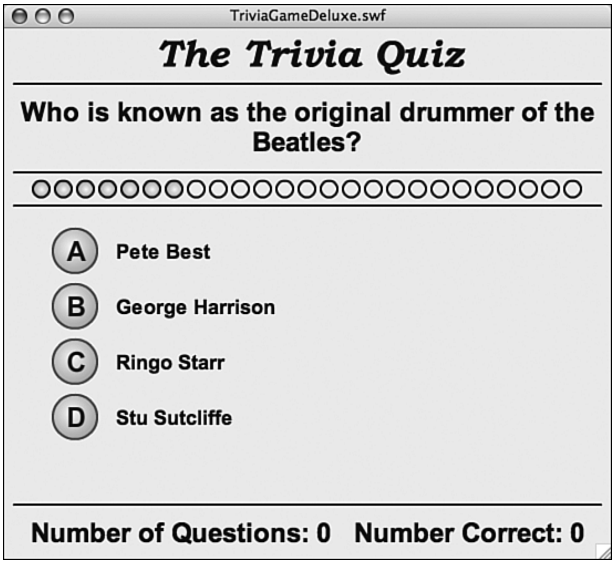

TriviaGameDeluxe.swf
The Trivia Quiz
Who is known as the original drummer of the Beatles?
A Pete Best
B George Harrison
C Ringo Starr
D Stu Sutcliffe
Number of Questions: 0 Number Correct: 0

Figure 10.6
The `Clock` has been added, and there is room for more features below.

NOTE

The use of 25 dots as a clock is completely arbitrary. You could make any 26-frame sequence as a movie clip and use that (a stopwatch or a progress bar, for example). You don't even need to use 25 separate elements. You could easily substitute five changes and spread the frames along the timeline.

Every second, the `updateClock` function is called. The `Clock` movie clip moves over one more frame. When the time is up, a message is displayed and `finishQuestion` is called just like it is when the player clicks an answer:

```
// update the clock
public function updateClock(event:TimerEvent) {
     clock.gotoAndStop(event.target.currentCount+1);
     if (event.target.currentCount == event.target.repeatCount) {
          messageField = createText("Out of time! The correct answer was:",
                    questionFormat,gameSprite,0,190,550);
          finishQuestion();
     }
}
```

Now the player has two ways to get a question wrong: clicking a wrong answer or letting the time expire.

Adding Hints

You might have noticed that the XML sample files include both a hint and an extra fact for all questions. We'll finally make use of one of them now.

To add simple hints to the game, we just include a "Hint" button next to each question. When the player clicks it, the button is replaced with the text hint.

Implementing this requires a few new things. First, we'll add a `hintFormat` to the class's definitions, along with the text variable definitions:

```
private var hintFormat:TextFormat;
```

Then, we'll set this format in the construction function:

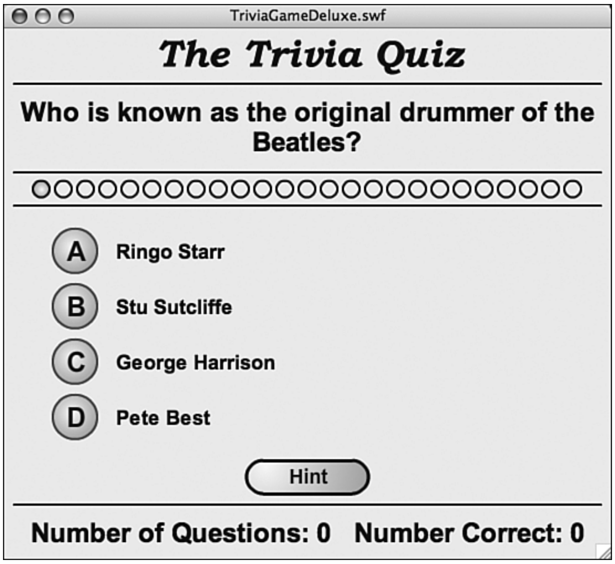
```
hintFormat = new TextFormat("Arial",14,0x330000,true,false,false,null,null,"center");
```

We'll also add a `hintButton` to the list of class's variables, along with the sprites and objects definitions:

```
private var hintButton:GameButton;
```

In the `askQuestion` function, we'll create the new "Hint" button and position it under the last answer, as shown in Figure 10.7:

```
// place the hint button
hintButton = new GameButton();
hintButton.label.text = "Hint";
hintButton.x = 220;
hintButton.y = 390;
gameSprite.addChild(hintButton);
hintButton.addEventListener(MouseEvent.CLICK,pressedHintButton);
```


TriviaGameDeluxe.swf

The Trivia Quiz

Who is known as the original drummer of the Beatles?

A Ringo Starr
B Stu Sutcliffe
C George Harrison
D Pete Best

Hint

Number of Questions: 0   Number Correct: 0

Figure 10.7
The "Hint" button appears near the bottom.

When the player clicks the button, it is removed. In its place will be a new text field, set to the small text format of `hintFormat`:

```
// player wants a hint
public function pressedHintButton(event:MouseEvent) {
     // remove button
     gameSprite.removeChild(hintButton);
     hintButton = null;

     // show hint
     var hint:String = dataXML.item[questionNum].hint;
     var hintField:TextField = createText(hint,hintFormat,questionSprite,0,390,550);
}
```

We also want to use the `removeChild` statement inside the `finishQuestion` function, checking first that the `hintButton` exists in case it was removed when the player clicked it:

```
//remove hint button
if (hintButton != null) {
     gameSprite.removeChild(hintButton);
}
```

This prevents the player from clicking the button after the question has already been answered.

That's all that we need to do to show the hint. Because the `hintField` is part of the `questionSprite`, it gets cleaned up when we remove that sprite when the player has finished. Figure 10.8 shows how the hint appears after the player clicks the button.

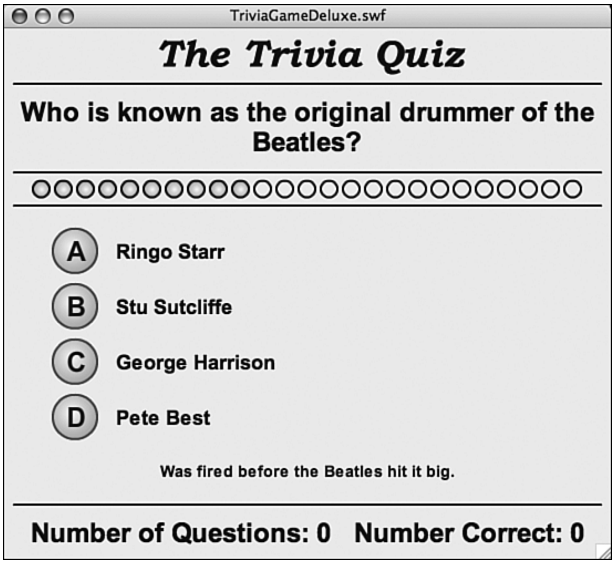


Figure 10.8
The hint appears in place of the button.

NOTE

What makes a good hint? Writing hints can be harder than writing the question and the answers. You don't want to give away the answer, but at the same time you want to help the player. Often the best way to go is to give a hint that points to the answer, but in a different context. For instance, if the question is about state capitals and the answer is Carson City, a hint might be "Also the name of a long-time host of the *Tonight Show.*"

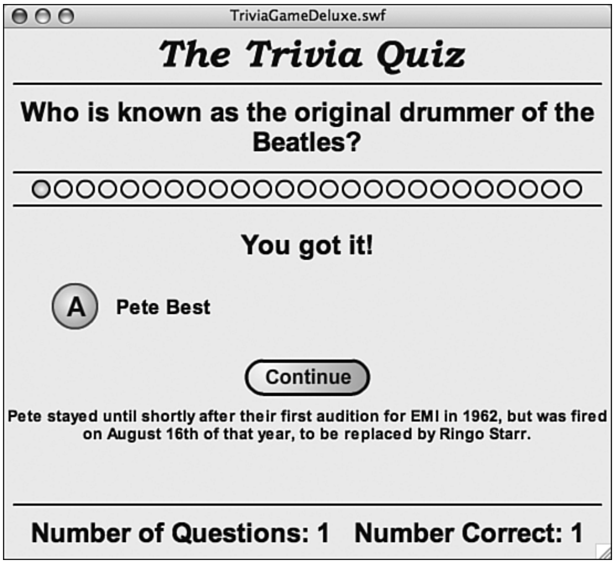
Adding a Factoid

Adding an extra fact, sometimes called a *factoid,* to the end of a question is relatively simple. It is similar to the hint functionality, but a factoid will automatically show up when the question has been answered.

No new variables are need for this. In fact, all that is needed is for a text field to be created and populated when the question is finished. This code is added to `finishQuestion`:

```
// display factoid
var fact:String = dataXML.item[questionNum].fact;
var factField:TextField = createText(fact,hintFormat,questionSprite,0,340,550);
```

Because the new `TextField` is part of `questionSprite`, it is disposed of at the same time. We are also using the `hintFormat` instead of creating a separate format for the factoid. Figure 10.9 shows the result.


TriviaGameDeluxe.swf

The Trivia Quiz

Who is known as the original drummer of the Beatles?

You got it!

A Pete Best

Continue

Pete stayed until shortly after their first audition for EMI in 1962, but was fired on August 16th of that year, to be replaced by Ringo Starr.

Number of Questions: 1 Number Correct: 1

Figure 10.9
The factoid displays when the player has answered the question.

When deciding on the location of the factoid, I took care to make sure the hint and the factoid can coexist. If the player chooses to see the hint, it will remain on screen after the player answers the question, and is displayed right below the factoid.

Adding Complex Scoring

The problem with the hint function, as well as with the clock, is that the player gets very little penalty when using the hint, or letting the time run long.

What makes the game more challenging is to have a score penalty for using the hint. In addition, we can have the total points scored dependent on how fast the player answers a question.

To make these changes, let's introduce two new variables. They can be placed anywhere in the variable definitions, though they fit best along with the existing game state variable definitions. These will keep track of the number of points the current question is worth and the total number of points the player has scored in the game so far:

```
private var questionPoints:int;
private var gameScore:int;
```

In the `startTriviaGame` function, we'll initialize the `gameScore` to 0, just before calling `showScore`:

```
gameScore = 0;
```

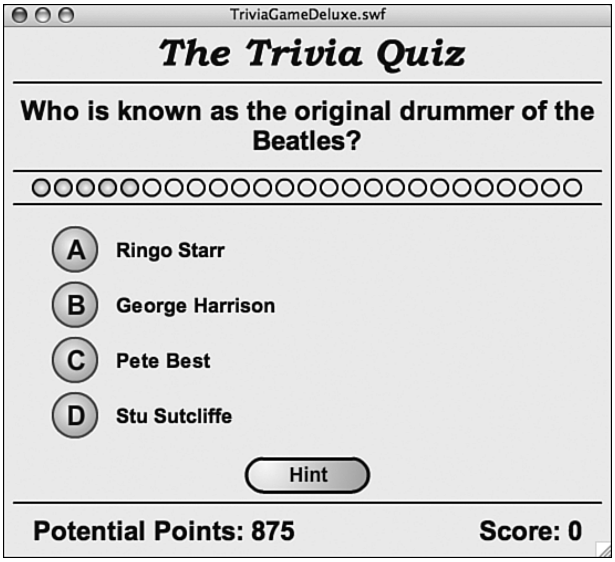
The `showScore` function will be replaced by a new version. This will show the number of points the question is worth and the player's current score:

```
public function showScore() {
     if (questionPoints != 0) {
          scoreField.text =
           "Potential Points: "+questionPoints+"\t   Score: "+gameScore;
     } else {
          scoreField.text =
           "Potential Points: ---\t   Score: "+gameScore;
     }
}
```

NOTE

The `\t` in the value for `scoreField.text` represents a tab character. By placing a tab between the two parts of the field, we make it possible for the text to remain in the same general space, even while the length of the numbers change. It is not a perfect solution, but is much simpler than creating two separate fields in this case. You might want to have two separate fields if you need more control over the placement of these numbers.

Figure 10.10 shows the new way the score is displayed at the bottom of the screen.


TriviaGameDeluxe.swf

The Trivia Quiz

Who is known as the original drummer of the Beatles?

A Ringo Starr

B George Harrison

C Pete Best

D Stu Sutcliffe

Hint

Potential Points: 875 Score: 0

Figure 10.10 *The number of questions asked and number correct has been replaced with the number of potential points for this question and the player's score.*

With the `showScore` function now working to update the potential points as well as the total score, we need to call it more often. Every time the `questionScore` changes, we need to call `showScore` to let the player know the new value.

If the `questionScore` is 0, we'll display `---` rather than 0. This will make it clearer that the potential points don't mean anything between questions.

In `askQuestion`, we'll set potential score for the question to 1,000:

```
// start question points at max
questionPoints = 1000;
showScore();

)
```

Then, for every second that goes by, we'll decrease the score. This happens in the `updateClock` function. Each time a new circle is filled in, 25 points are removed from the potential score:

```
// reduce points
questionPoints -= 25;
showScore();
```

Also, the potential points decrease when player requests a hint. That will cost them 300 points:

```
// penalty
questionPoints -= 300;
showScore();
```

Of course, the only way the user gets any points is by guessing the right answer. So, this will be added in the appropriate place in `clickAnswer`:

```
gameScore += questionPoints;
```

No need to call `showScore` here because it will be called immediately after in the `finishQuestion` function. In fact, here's where we'll be setting `questionPoints` to 0, too:

```
questionPoints = 0;
showScore();
```

You can also opt to keep the original score text field at the bottom and display the potential points and score in a separate field. Then, players can see all the statistics on how they are doing.

The movies **`TriviaGameDeluxe.fla`** and **`TriviaGameDeluxe.as`** leave in the `numCorrect` and `numQuestionsAsked` for this purpose, even though they don't use them.

Randomizing the Questions

You may or may not want your trivia quiz game to present the same questions each time someone plays. It depends on how you are using the game.

If you want to present different questions each time, and your game is a web-based game, it is ideal to have a server-based application that creates a random XML document of trivia questions from a large database.

However, if your needs are simpler, but you still want a number of random questions chosen from a relatively small total number of questions, there is a way to do it completely within ActionScript.

After the XML document is read in, this raw data can be processed into a smaller XML document with a set number of random questions.

The new beginning of the `xmlLoaded` function would look like this:

```
public function xmlLoaded(event:Event) {
     var tempXML:XML = XML(event.target.data);
     dataXML = selectQuestions(tempXML,10);
```

The `selectQuestions` function takes the complete data set, plus a number of questions to return. This function picks random `item` nodes from the original XML document and creates a new XML object:

```
// select a number of random questions
public function selectQuestions(allXML:XML, numToChoose:int):XML {

     // create a new XML object to hold the questions
     var chosenXML:XML = <trivia></trivia>;
```

```
        // loop until we have enough
        while(chosenXML.child("*").length() < numToChoose) {

            // pick a random question and move it over
            var r:int = Math.floor(Math.random()*allXML.child("*").length());
            chosenXML.appendChild(allXML.item[r].copy());

            // don't use it again
            delete allXML.item[r];
        }

        // ret
        return chosenXML;
    }
```

This random selection and shuffle of questions is very handy for creating a quick solution. However, if you have more than 100 questions, for example, it is important that you don't require the movie to read in such a large XML document each time. I recommend a server-side solution. If you don't have server-side programming experience, you probably want to team up with, or hire someone, who does.

Picture Quiz

SOURCE FILES
http://flashgameu.com
A3GPU10_PictureTriviaGame.zip

Not all question and answer games work well with just text. Sometimes a picture represents an idea better. For instance, if you want to test someone's geometry knowledge, text questions and answers are not always going to be able to convey the idea you want to test.

Converting our simple trivia game engine to something that uses images is actually not that hard. We just need to rearrange the screen a bit, and then allow for the loading of some external image files. The main part of the quiz engine can remain the same.

Better Answer Arrangement

Before we can load images, we need a better arrangement for the answers on the screen. Figure 10.11 shows the answers in a 2x2 formation, rather than four lines.

This provides a better space for images of approximately 250 pixels wide by 100 pixels high, at most. Probably best to stick with 200 by 80 so that the loaded images don't intrude on the other buttons.

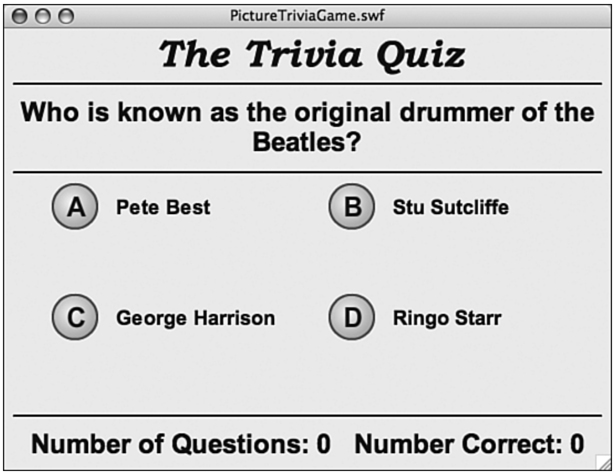

PictureTriviaGame.swf
The Trivia Quiz
Who is known as the original drummer of the Beatles?
A Pete Best
B Stu Sutcliffe
C George Harrison
D Ringo Starr
Number of Questions: 0 Number Correct: 0

Figure 10.11
The answers are now stacked in two columns and two rows.

Getting this arrangement is just a matter of a change to the middle of `askQuestion`. The variables `xpos` and `ypos` keep track of the current position and start at 0 and 0. Then, 1 is added to `xpos` to move over to the right. After that, `xpos` is set back to 0, and `ypos` is increased. This places the four answers at positions 0,0, 1,0, 0,1, and 1,1. This corresponds to screen locations 100,150, 350,150, 100,250, and 350,250:

NOTE
We'll be making more changes to this section of code in a bit, so the following code will not match the final **PictureTriviaGame.as** yet, in case you are following along.

```
// put each answer into a new sprite with a circle icon
answerSprites = new Sprite();
var xpos:int = 0;
var ypos:int = 0;
for(var i:int=0;i<answers.length;i++) {
     var answer:String = answers[i];
     var answerSprite:Sprite = new Sprite();
     var letter:String = String.fromCharCode(65+i); // A-D
     var answerField:TextField = createText(answer,answerFormat,answerSprite,0,0,200);
     var circle:Circle = new Circle(); // from Library
     circle.letter.text = letter;
     answerSprite.x = 100+xpos*250;
     answerSprite.y = 150+ypos*100;
     xpos++
     if (xpos > 1) {
          xpos = 0;
          ypos += 1;
```

```
        }
        answerSprite.addChild(circle);
        answerSprite.addEventListener(MouseEvent.CLICK,clickAnswer); // make it a button
        answerSprites.addChild(answerSprite);
    }
```

This is already a useful modification because it presents the answers is a more interesting way than just four straight down.

Recognizing Two Types of Answers

The goal here is not to create a quiz that only takes images as answers, but one that allows you to mix up text and images. So, we need to be able to specify in the XML file what type an answer is. We can do this by adding an attribute to the answer in the XML:

```
<item>
    <question type="text">Which one is an equilateral triangle?</question>
    <answers>
        <answer type="file">equilateral.swf</answer>
        <answer type="file">right.swf</answer>
        <answer type="file">isosceles.swf</answer>
        <answer type="file">scalene.swf</answer>
    </answers>
</item>
```

To determine whether an answer should be displayed as text, or an external file loaded, we just look at the `type` property. Next, we'll modify our code to do this.

Creating Loader Objects

In `shuffleAnswers`, we build a randomly sorted array of answers from the ones in the XML object. The text of these answers is stored in an array. However, now we need to store both the text and the type of these answers. So, the line where we add a new answer to the array changes to this:

```
shuffledAnswers.push({type: answers.answer[r].@type, value: answers.answer[r]});
```

Now, when we create each answer, we need to determine whether the answer is text or an image. If it is an image, we'll create a `Loader` object. This is like a movie clip taken from the library, except that you use a `URLRequest` and the `load` command to retrieve the movie clip contents from an external file:

```
var answerSprite:Sprite = new Sprite();
if (answers[i].type == "text") {
    var answerField:TextField =
         createText(answers[i].value,answerFormat,answerSprite,0,0,200);
} else {
    var answerLoader:Loader = new Loader();
```

```
        var answerRequest:URLRequest = new URLRequest("triviaimages/"+answers[i].value);
        answerLoader.load(answerRequest);
        answerSprite.addChild(answerLoader);
    }
```

The code assumes that all the images are inside a folder named triviaimages.

`Loader` objects can act autonomously. After you set them into action with the `load` command, they get the file from the server and appear at their designated position when they are ready. You don't need to track them or do anything when the loading is complete.

NOTE

If you are combining this example with the `Clock` function of the previous movie, you'll want to do some extra work. Is it fair for the clock to be counting down if some of the answers have not yet appeared? So, you want to listen for the `Event.COMPLETE` for each `Loader` and only start the clock after all the answers have displayed.

Figure 10.12 shows the quiz with four external movies loaded into the answers.

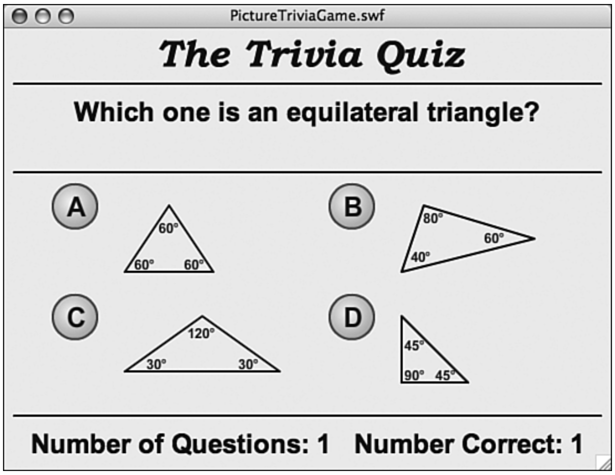


Figure 10.12
External movies have replaced text in each answer.

Determining the Right Answer

We previously relied on the `text` property of the answer field to determine whether the player got the answer right. We can't do that anymore because the `Loader` object movie clip does not have a `text` property like a `TextField` does. So, instead, we'll take advantage of the fact that the second object in the answerSprite is the dynamically created Circle. We can attach an answer property to that and store the answer there:

```
circle.answer = answers[i].value;
```

Then, in the clickAnswer function, we'll look at this new answer property to determine whether the player clicked the right sprite:

```
var selectedAnswer = event.currentTarget.getChildAt(1).answer;
```

Note that the Circle is child number 1 in the answerSprite. Previously, we were looking at child number 0, which was the TextField.

Another change that is needed is to properly set the position of the correct answer when the player has made his or her choice. Previously, the answers were all in a single-file column, with the same x value. So when we wanted to center the correct answer, we just set the y value of the correct answerSprite. But now that the answer can be on the left or the right, we want to set the x value, too. Here is the new code for the finishQuestion function:

```
answerSprites.getChildAt(i).x = 100;
answerSprites.getChildAt(i).y = 200;
```

Expanding the Click Area

One last item before we are done with the answers. If the answer is text, players can click either the Circle or the TextField to register their answer. However, with loaded movies as answers, they may not have much to click on. In Figure 10.11, the answers are just some narrow lines that make up a triangle.

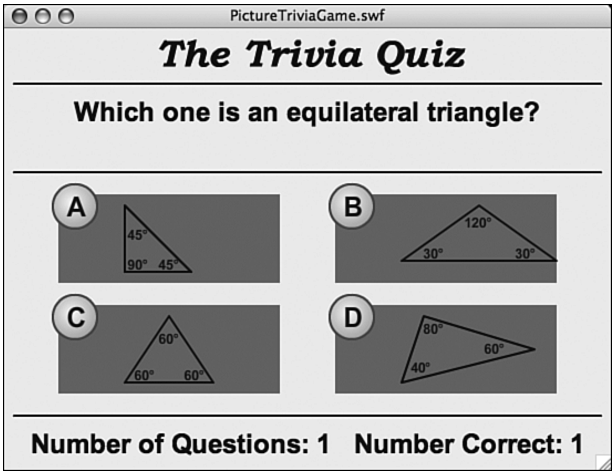
So to click the answer, players must either click the Circle or click some part of the graphic. However, they should be able to click any reasonable part of the answer.

A quick way to fix this is to place a solid rectangle inside of each answer sprite. They can be done by just drawing with a solid color, but a 0 setting for the alpha channel to make it invisible:

```
// set a larger click area
answerSprite.graphics.beginFill(0xFFFFFF,0);
answerSprite.graphics.drawRect(-50, 0, 200, 80);
```

Figure 10.13 shows the answer sprites with a graphic behind each one. Instead of an alpha value of 0, I've change the alpha to .5 so that you can see the rectangle.

Figure 10.13
A rectangle has been drawn behind each answer.



Now players can click in the general area of each answer.

Images for Questions

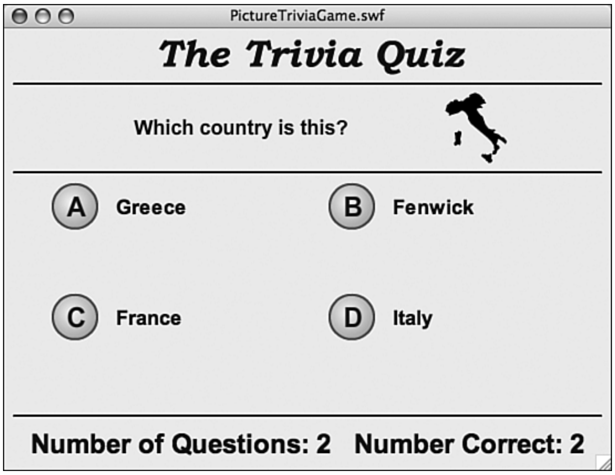
In addition to using images in the answers, you might want to use images for the question itself. We'll do this in the same way, by using a `type` attribute in the XML:

```
<item>
    <question type="file">italy.swf</question>
    <answers>
        <answer type="text">Italy</answer>
        <answer type="text">France</answer>
        <answer type="text">Greece</answer>
        <answer type="text">Fenwick</answer>
    </answers>
</item>
```

Adding this to our ActionScript code is easier this time because the question is not an active element. We just need to use a `Loader` object rather than a `TextField`. Here is the change to `askQuestion`:

```
//create text field for question
var question:String = dataXML.item[questionNum].question;
if (dataXML.item[questionNum].question.@type == "text") {
    questionField = createText(question,questionFormat,questionSprite,0,60,550);
} else {
    var questionLoader:Loader = new Loader();
    var questionRequest:URLRequest = new URLRequest("triviaimages/"+question);
    questionLoader.load(questionRequest);
    questionLoader.y = 50;
    questionSprite.addChild(questionLoader);
}
```

Figure 10.14 shows a question that uses an external image as a question and four text answers. You could, of course, have a question and all four answers that are external files.



Which country is this?

Greece

Fenwick

France

Italy

Number of Questions: 2 Number Correct: 2

Figure 10.14
The question is an external Flash movie, but the four answers are just text.

Figure 10.14 also demonstrates how using external files for questions and answers doesn't mean they have to be drawings or pictures. They can also include text. This could come in handy for a math quiz that needs to use complex notation such as fractions, exponentials, or symbols.

Modifying the Game

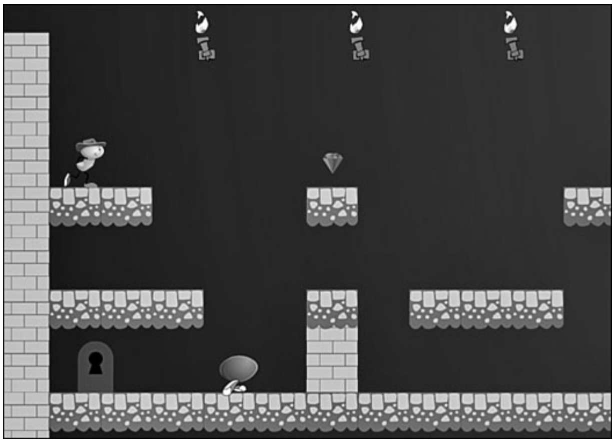
Trivia games are only as good as the questions and answers in them, no matter how well-designed the program and interface are. If you plan on making a game for entertainment, then you need to have questions that are engaging and answers that are as well. If you are making a game for educational purposes, then you need to make sure the questions and answers are clear and fair.

You can modify this game to have fewer or more answers relatively easy. If you want, you can have just two answers, such as True and False. Rarely are more than four answers seen, although you sometimes see "All of the above" or "None of the above." No special programming is needed for these, except possibly an exception to keep these as the fifth and/or sixth answer in the list.

Beyond the questions and answers and how they are displayed, one modification is to have a game metaphor. This would be a visual representation of how a player is doing in the game. It can also modify how the game plays.

For instance, the player can have a character that is climbing a rope. For every answer they get right, the character moves up the rope. When the player gets one question wrong, the character falls down to the bottom. The object is to get to the top by correctly answering a number of questions consecutively.

Game metaphors can be used to tie the game closer to the website or product it's part of. For instance, a wildlife conservation site could have a trivia game using animals.



11

Action Games: Platform Games

- Designing the Game
- Building the Class
- Modifying the Game

SOURCE FILES
http://flashgameu.com
A3GPU11_PlatformGame.zip

Side-scrolling games, also called platform games, first appeared in the early 1980s, and soon became the standard way to produce video games until 3D took over in the 1990s.

Side-scrolling platform games allow the player to control a character who is shown in side view. He can move left and right, and usually can jump. As he moves off the screen to one side, the background scrolls to reveal more of the play area.

NOTE
Platform games almost always feature a jumping character. The most famous is, of course, Nintendo's Mario. He appeared in dozens of games, from Donkey Kong to numerous adventures on Nintendo's consoles.

In this chapter, we build a fairly simple side-scrolling platform game with a main character that can move left and right and can jump. There will be walls and platforms. There will also be enemies, and items to collect.

Designing the Game

Before we begin programming, let's think through all the aspects of the game. We need a hero, enemies, items, and a way to build levels.

Level Design

A very important aspect of platform games is level design. Like the trivia game in the preceding chapter can't exist without content, a platform game needs content, too. The content in this case is level design.

Someone—the programmer, an artist, or a level designer—needs to build levels. Figure 11.1 shows such a level, in fact the first level for our game.

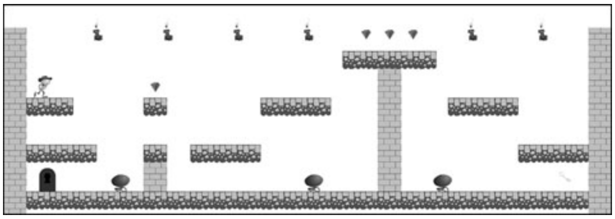


Figure 11.1
Level 1 of our platform game features three enemies, several treasures, a key and a door.

More sophisticated platform games might also have a level editor. This would allow a level designer to create many different levels and try them out, while programmers and artists work on the rest of the game.

For our simple example, however, we'll design the levels right in Flash, by building a movie clip containing the various game pieces.

The movie **`PlatformGame.fla`** contains various game pieces in its library. Here is a list:

- **Floor**—A simple 40x40 block that the hero can stand on. It will block the hero on the left and right side.
- **Wall**—Identical to the floor in use, but visually different.
- **Hero**—The player's character. Includes stand, walk, jump, and die animations.
- **Enemy**—A simpler character with a walk animation.
- **Treasure**—A simple jewel, with animated sparkles.
- **Key**—A simple key graphic.
- **Door**—A door, with an open animation.

To create a level from these objects, they simply need to be dragged around and placed within a movie clip.

Figure 11.1 showed such a movie clip. It is actually the movie clip `gamelevel1` in the library of the movie **`PlatformGame.fla`**.

Wall and Floor Pieces

To make it easy to build this level, I set a grid up using View, Grid, Edit Grid set to 40 by 40. Then, I dragged floor and wall pieces from the library and placed them along this grid.

Figure 11.2 shows the level with the background hidden so that you can see the grid marks.

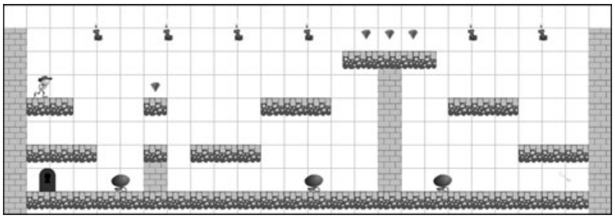
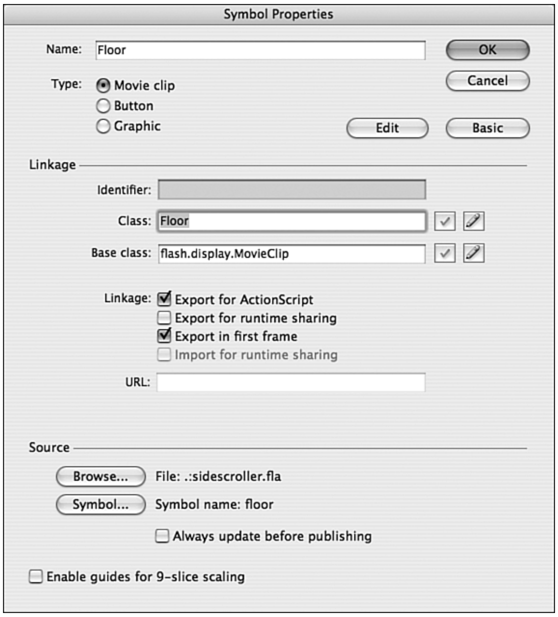


Figure 11.2
Placing the wall and floor pieces along the grid makes it much easier to set up a level.

The wall and floor pieces in this game differ visually, but will both be seen as the same type of object in the code. You can add more "building block" pieces like this to make the game more diverse.

NOTE
An idea not used in this game is to have many different versions of the wall and floor blocks, all stored in different frames of the movie clip. Then, at the start of the game, a random frame is chosen for each block.

The walls and floor don't need to be named anything special. However, the library elements are set to Export for ActionScript, as you can see in Figure 11.3. This is vitally important because our code will be looking for `Floor` and `Wall` objects.



Symbol Properties

Name: Floor

Type: [✓ Movie clip] Button Graphic

Linkage

Identifier:

Class: Floor

Base class: flash.display.MovieClip

Linkage: [✓ Export for ActionScript] Export for runtime sharing [✓ Export in first frame] Import for runtime sharing

URL:

Source

File: .:sidescroller.fla

Symbol name: floor

Always update before publishing

Enable guides for 9-slice scaling

Figure 11.3
The `Floor` movie clip is set to also be the class object `Floor`.

The Hero and Enemies

The hero is seen in Figures 11.1 and 11.2 at the upper left. He has been placed carefully so that he is standing right on a `Floor` block.

The hero's movie clip has several different animated sections. Take a look at Figure 11.4.

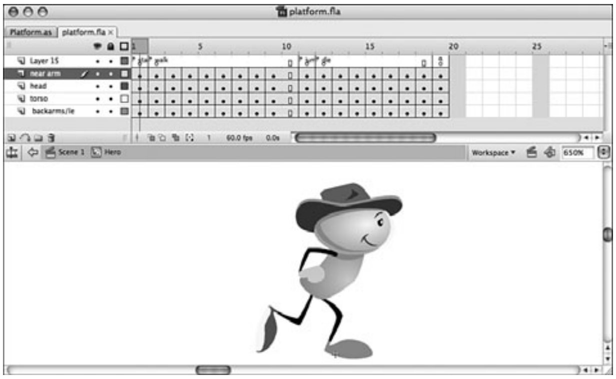


Figure 11.4
The hero has frames for stand, walk, jump, and die.

Notice that the registration point for the movie clip rests at the very bottom of the character's feet. We'll be matching this up with the top of the `Floor` piece that the character is resting on. Horizontally, the character is centered.

When placing the hero, you'll want the `y` position to match the `y` position of the `Floor` right underneath him. This way, the hero will start off standing. If you place him above a `Floor` block, he'll start by falling onto the `Floor`. This is another way to go, as you may envision the character falling into the scene.

The enemy is a similar movie clip, but with just a stand and walk sequence. Figure 11.5 shows him. Note that the registration point for the enemy is also at the bottom of his foot.

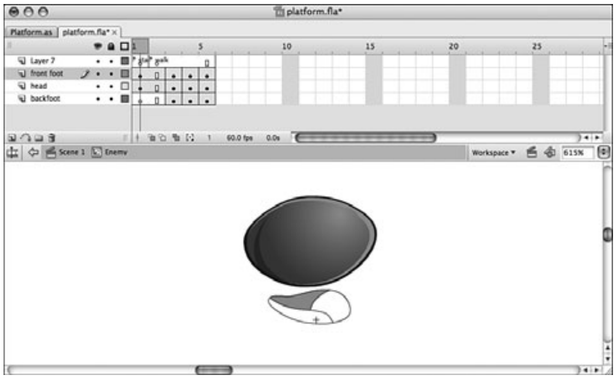


Figure 11.5
The enemy character has just stand and walk labels.

These little guys are designed to be jumped on. In fact, that's how the hero will get rid of them. So, they are short and squat. We don't need a "die" sequence because we'll be removing them from the screen the instant that they are destroyed, and using the `PointBurst` from Chapter 8, "Casual Games: Match Three," to show a message in their place.

The enemies should also be placed directly matching a `Floor` piece. If not, they will drop down to the next `Floor`, which works out fine if that is how you want them to start.

The enemies also need to have instance names. The three shown in the Figures 11.1 and 11.2 are named enemy1, enemy2, and enemy3.

Another thing about enemies: They will be programmed to walk back and forth, turning around when they hit walls. So, you should place them in an area with walls on both sides. If they have a drop off to one side, they will fall down that drop off at the first opportunity. They will keep dropping down every time they get the opportunity, until they stabilize going back and forth in an area with walls on either end.

Treasures and Items

In Figures 11.1 and 11.2, you can see various other objects. The ruby-looking items are treasures that can be obtained for points. There is nothing remarkable about the `Treasure` movie clip, other than the fact that it has many frames to accommodate a looping sparkle animation. This does not affect the game in any way; it is just for visual interest.

NOTE

If you want more than one type of treasure, the easy way to accomplish it is to place several treasures, one per frame, in that one movie clip. Otherwise, you must create a variety of different objects, such as `Diamond`, `Coin`, `TheOneTrueRing`, and so on. Then, you need to look for each one in the code.

The `Key` and `Door` movie clips are similar. The `Key` has some frames for animation only. The `Door`, on the other hand, has an "open" sequence starting on frame 2. Figure 11.6 shows this movie clip.

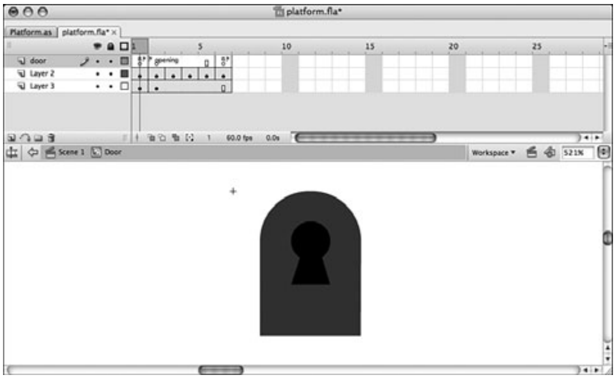


Figure 11.6
The `Door` *movie clip has a static frame 1, and then a short animated sequence of it opening.*

The items don't need to be placed perfectly on the grid. They just need to be put in reach of the hero as he walks by, or in other cases, jumps. It does help visually if the door is resting on the ground, however.

NOTE

Pay attention to the layering of your game elements. This will be maintained while the game is being played. So, for instance, if the hero is *behind* a wall or other object, the wall appears in front of the hero graphic when the hero is close to it.

On the other hand, you can have objects appear in front of the player, like a semi-transparent pane of glass or a small piece of furniture.

The `Treasure`, `Key`, and `Door` movie clips are all set to Export for ActionScript with class names to match their library names. Our code will be looking for them by class. The movie clip instances themselves don't need to have any names.

One other item in the game is the `Chest`. This is a two-frame movie clip of a treasure chest closed, then open. It is the object of the player's quest, and the game will end when the player finds it.

Background Art

The game levels also include a layer in the movie clip with background art. In this case, it is a shaded gradient rectangle. However, there could be more. In fact, anything you add to the background will be visible, yet not active.

So, you can color in the scene however you want. There can be pictures hanging on the walls, animated torches, messages, and signs.

Figures 11.1 and 11.2 show torches hanging up high at the top. These are just placed on the same background layer as the gradient. Our game code won't even need to acknowledge these because they will just scroll along with the background.

The Dialog Box

This movie will include a dialog box that we can bring up at any time to convey some information to players and await their input. You can see the dialog box movie clip in Figure 11.7.

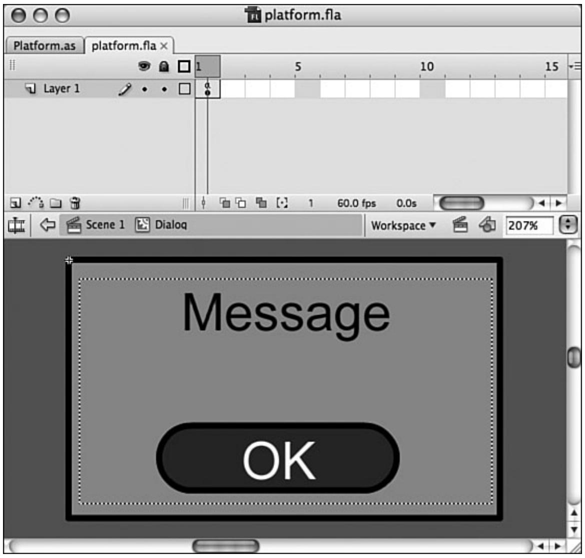


Figure 11.7
The `Dialog` *movie clip will be used to wait for the player to click a button before continuing.*

The dialog box will be displayed when the player dies, the game is over, a level is complete, or the player wins. When any of these events happen, the game halts and a dialog box displays. We'll include the dialog box as a separate movie clip, complete with a dynamic text field and a button.

The Main Timeline

The main timeline features a "start" frame with instructions. After that, each frame contains one game level movie clip. This makes it easy to jump from level to level, both while the game is playing and while you are creating the levels.

The second frame contains `GameLevel1`, which has an instance name of simply `gamelevel`. The third frame contains `GameLevel2`, which also has an instance name of `gamelevel`.

When the ActionScript executes, it looks for the movie clip with the instance name `gamelevel` on the current frame. This enables us to place different game level movie clips on different frames.

On the game level frames, we'll have three dynamic text fields: one for the level, one for the number of lives remaining, and one for the score. Figure 11.8 shows what the screen looks like after the game begins.

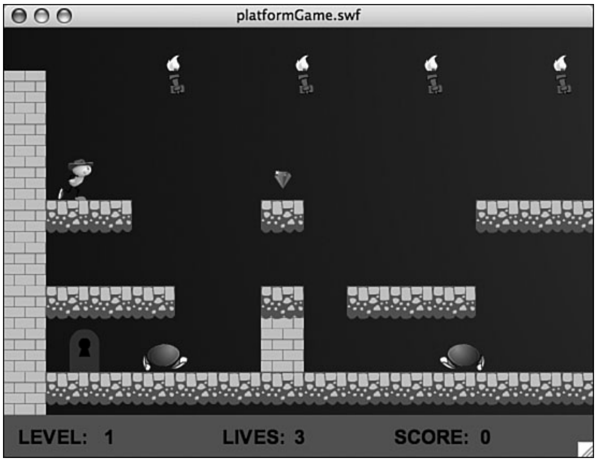


Figure 11.8
There are three text fields at the bottom of the screen.

Designing the Class

The class starts by examining the `gamelevel` movie clip. It loops through each of the children in this movie clip and figures out what it does and how it needs to be represented in the game class.

For instance, if a child is a `Wall` or `Floor`, it is added to an array of such objects. Then, when the characters move around, these objects are checked for collisions.

The hero and the enemies are also looked for. It is assumed that the hero has an instance name of `hero`, and the enemies are named `enemy1`, `enemy2`, and so on.

NOTE

To determine what type of object a movie clip is, we'll be using the `is` operator. This operator compares the object type of a variable against an object type (for instance `(ThisThing is MyObject)`).

The largest part of the code, by far, is the part that deals with movement. The hero can move left, right, and he can jump. But, he is also affected by gravity and can fall off of edges. He can collide with walls and be stopped, and also collides with floors, which prevent him from falling through them.

The enemies do the same thing, except that their movements are not affected by the arrow keys. But, they still follow the same rules as the hero.

So, instead of having the hero and the enemies use different movement code, we'll have them share a single character movement function.

Horizontal scrolling is another movement factor. The hero and enemies will move inside the `gamelevel` movie clip. If the hero's relative position on the stage goes too far to the left or right, however, we'll move the entire `gamelevel` movie clip to make it scroll. The rest of the code can ignore this because nothing will actually move inside the `gamelevel`.

Planning Which Functions Are Needed

Before we begin programming, let's take a look at all of the functions that we will be using in the class, and which ones will rely on each other.

- **`startPlatformGame`**—Initializes the score and player lives.
- **`startGameLevel`**—Initializes the level, calling the next three functions.
    - **`createHero`**—Creates the hero object, looking at the placement of the `hero` movie clip instance.
    - **`addEnemies`**—Creates the enemy objects, looking at the `enemyX` movie clips.
    - **`examineLevel`**—Looks for walls, floors, and other items in the `gamelevel` movie clip.
- **`keyDownFunction`**—Notes key presses by the user.
- **`keyUpFunction`**—Notes when the user is done pressing a key.
- **`gameLoop`**—Called every frame to calculate the time passed and then call the next four functions.
    - **`moveEnemies`**—Loops through all enemies and moves them.
    - **`moveCharacter`**—Moves the character.
    - **`scrollWithHero`**—Scrolls the `gamelevel` movie clip depending on the location of the hero.

- **`checkCollisions`**—Check to see whether the hero hit any enemies or items. Calls the next three functions.
  - **`enemyDie`**—Enemy character is removed.
  - **`heroDie`**—Hero loses a life, game possibly over.
  - **`getObject`**—Hero gets an item.
- **`addScore`**—Add points to the score, display the score.
- **`showLives`**—Show the number of lives left.
- **`levelComplete`**—Level is done, pause and display dialog.
- **`gameComplete`**—Treasure is found, pause and display dialog.
- **`clickDialogButton`**—Dialog button clicked, perform next action.
- **`cleanUp`**—Remove the `gamelist` to prepare for the next level.

Now that we know the functions we need to write, let's build the **PlatformGame.as** class.

Building the Class

The package file is not particularly long, especially considering all that this game does. Because of that, we'll keep everything in one class, even though it would be useful for a larger game to have separate classes for characters, items, and fixed objects.

Class Definition

At the start of the class, we can see our standard `import` listing, including the `flash.utils.getTimer` that we need for time-based animation:

```
package {
     import flash.display.*;
     import flash.events.*;
     import flash.text.*;
     import flash.utils.getTimer;
```

We need only a few constants. The first is `gravity`, and then also the distance to the edges of the screen is needed to start horizontal scrolling.

NOTE

The `gravity` constant was achieved through trial and error. Knowing that it would be multiplied by the number of milliseconds between steps, I started with a very low fraction. Then, I adjusted it after the game was complete, until I had the jump and fall behavior that seemed right.

```
    public class PlatformGame extends MovieClip {
        // movement constants
        static const gravity:Number = .004;

        // edge for scrolling
        static const edgeDistance:Number = 100;
```

When the gamelevel is scanned, all the objects found are placed in one of two arrays. The fixedObjects array holds references to any objects that the player can stand on or be blocked by. The otherObjects array holds items like the Key, Door, Chest, and Treasure:

```
        // object arrays
        private var fixedObjects:Array;
        private var otherObjects:Array;
```

The hero movie clip is already named "hero" and can be accessed through gamelevel.hero. But the hero object in our class holds that reference, and many other pieces of information about the hero character. Similarly, the enemies array holds a list of objects with information about each enemy:

```
        // hero and enemies
        private var hero:Object;
        private var enemies:Array;
```

A number of variables are needed to keep track of the game state. We'll use playerObjects as an array to store objects that the player has picked up. The only one in the game is the Key, but we'll store it in an array anyway to pave the way for more objects to be added.

The gameMode is a string that will help convey to various functions what has happened to the hero. It will start with a value of "start" and then get changed to "play" when the game is ready to go.

The gameScore and playerLives correspond to the number of points scored and the number of lives remaining for the player.

The lastTime variable holds the millisecond value of the last step of game animation. We'll be using it to drive the time-based animation used by game elements:

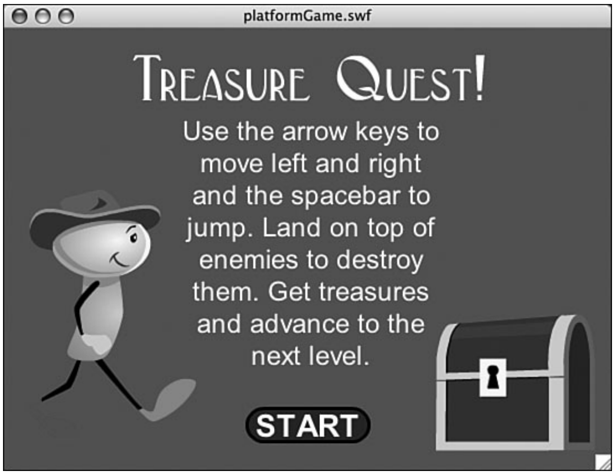
```
        // game state
        private var playerObjects:Array;
        private var gameMode:String = "start";
        private var gameScore:int;
        private var playerLives:int;
        private var lastTime:Number = 0;
```

Starting the Game and Level

When the game starts, we need to set some of the game state variables. This is done by calling the `startPlatformGame` function on the frame that contains the first game level. We'll have some other variables that need to be reset every level. Those are set when the `startGameLevel` is called on the next frame:

```
// start game
public function startPlatformGame() {
    playerObjects = new Array();
    gameScore = 0;
    gameMode = "play";
    playerLives = 3;
}
```

Figure 11.9 shows the game start screen, with a button that the player must press to continue.



Use the arrow keys to move left and right and the spacebar to jump. Land on top of enemies to destroy them. Get treasures and advance to the next level.

Figure 11.9
The start screen for the platform game.

The `startGameLevel` Function

The `startGameLevel` function is called on every frame that contains a `gamelevel` movie clip. It then delegates the tasks of finding and setting the hero, enemies, and game items:

```
// start level
public function startGameLevel() {

    // create characters
    createHero();
    addEnemies();

    // examine level and note all objects
    examineLevel();
```

The `startGameLevel` function also sets up three event listeners. The first is the main `gameLoop` function, which will execute each frame to push forward the animation. The other two are the keyboard event listeners we need to get player input:

```
        // add listeners
        this.addEventListener(Event.ENTER_FRAME,gameLoop);
        stage.addEventListener(KeyboardEvent.KEY_DOWN,keyDownFunction);
        stage.addEventListener(KeyboardEvent.KEY_UP,keyUpFunction);
```

Finally, the `gameMode` is set to `"play"`, and two functions are called to set up the display of the score and lives. The score display is updated with a call to `addScore`, which both adds a number of points to the score and updates the text field. So, if we add 0 points, it acts just like a display function:

```
        // set game state
        gameMode = "play";
        addScore(0);
        showLives();
    }
```

The `createHero` Function

The `hero` movie clip is already in the `gamelevel` movie clip and ready to go. But, we need to set and use many properties, so we'll create a `hero` object in the class to store these properties:

```
    // creates the hero object and sets all properties
    public function createHero() {
        hero = new Object();
```

The first property is a reference to the movie clip that is the visual representation of the hero. Now we can refer to the hero as `hero.mc` rather than `gamelevel.hero`. This will fit better when we are using the `hero` object for all our manipulations of the player's character:

```
        hero.mc = gamelevel.hero;
```

Next are two properties that describe the velocity of the `hero`:

```
        hero.dx = 0.0;
        hero.dy = 0.0;
```

The `hero.inAir` property will be set to `true` when the `hero` is not resting on solid ground:

```
        hero.inAir = false;
```

The `hero.direction` property will be either `-1` or `1`, depending on the direction the `hero` is facing:

```
        hero.direction = 1;
```

The `hero.animstate` property will hold either `"stand"` or `"walk"`. If it is `"walk"`, we'll know that the character should be moving long its walk sequence. The frames in this sequence are stored in `hero.walkAnimation`. In this case, the walk sequence is on frames 2 through 8. To keep track of which step in the animation is currently showing, we'll use `hero.animstep`:

```
    hero.animstate = "stand";
    hero.walkAnimation = new Array(2,3,4,5,6,7,8);
    hero.animstep = 0;
```

The `hero.jump` property is set to `true` when the player presses the spacebar. Similarly, the `hero.moveLeft` and `hero.moveRight` will toggle between `true` and `false` depending on whether the arrow keys are pressed:

```
    hero.jump = false;
    hero.moveLeft = false;
    hero.moveRight = false;
```

The next few properties are constants used to determine how high the character jumps and how fast the character walks:

```
    hero.jumpSpeed = .8;
    hero.walkSpeed = .15;
```

NOTE

These constants were also determined with trial and error. I started with educated guesses, such as the character should be able to walk about 100 to 200 pixels per second. Then I adjusted as I built the game.

The `hero.width` and `hero.height` constants are used when determining collisions. Instead of using the actual width and height of the character, which varies depending on which frame of animation is shown, we will use the following constants:

```
    hero.width = 20.0;
    hero.height = 40.0;
```

When the hero does have a collision, we'll be resetting him to his starting position in the level. So, we need to record this location for use at that point:

```
    hero.startx = hero.mc.x;
    hero.starty = hero.mc.y;
}
```

The addEnemies Function

The enemies are stored in objects that look just like the hero object. With the hero and enemy objects having the same properties, we can feed either one into the moveCharacter function.

The addEnemies function looks for a movie clip named enemy1 and adds it to the enemies array, as an object. It then looks for enemy2 and so on.

One of the few differences between enemies and heroes is that enemies don't need the startx and starty properties. Also, the enemy.moveRight property starts off as true, so the enemy starts by walking to the right:

```
// finds all enemies in the level and creates an object for each
public function addEnemies() {
    enemies = new Array();
    var i:int = 1;
    while (true) {
        if (gamelevel["enemy"+i] == null) break;
        var enemy = new Object();
        enemy.mc = gamelevel["enemy"+i];
        enemy.dx = 0.0;
        enemy.dy = 0.0;
        enemy.inAir = false;
        enemy.direction = 1;
        enemy.animstate = "stand"
        enemy.walkAnimation = new Array(2,3,4,5);
        enemy.animstep = 0;
        enemy.jump = false;
        enemy.moveRight = true;
        enemy.moveLeft = false;
        enemy.jumpSpeed = 1.0;
        enemy.walkSpeed = .08;
        enemy.width = 30.0;
        enemy.height = 30.0;
        enemies.push(enemy);
        i++;
    }
}
```

The examineLevel Function

After the hero and all the enemies have been found, the examineLevel function looks at all the children of the gamelevel movie clip:

```
// look at all level children and note walls, floors and items
public function examineLevel() {
    fixedObjects = new Array();
    otherObjects = new Array();
    for(var i:int=0;i<this.gamelevel.numChildren;i++) {
        var mc = this.gamelevel.getChildAt(i);
```

If the object is a `Floor` or `Wall`, it is added to the `fixedObjects` array as an object with a reference to the movie clip, but also some other information. The locations of all four sides are stored in `leftside`, `rightside`, `topside`, and `bottomside`. We need quick access to these when determining collisions:

```
        // add floors and walls to fixedObjects
        if ((mc is Floor) || (mc is Wall)) {
            var floorObject:Object = new Object();
            floorObject.mc = mc;
            floorObject.leftside = mc.x;
            floorObject.rightside = mc.x+mc.width;
            floorObject.topside = mc.y;
            floorObject.bottomside = mc.y+mc.height;
            fixedObjects.push(floorObject);
```

All other objects are added very simply to the `otherObjects` array:

```
        // add treasure, key and door to otherOjects
        } else if ((mc is Treasure) || (mc is Key) ||
                 (mc is Door) || (mc is Chest)) {
            otherObjects.push(mc);
        }
    }
}
```

Keyboard Input

Accepting keyboard input will work as it did in previous games, using the arrow keys. However, we'll directly set the `moveLeft`, `moveRight`, and `jump` properties of the `hero`. We'll only allow `jump` to go to `true` if the hero isn't already in the air:

```
// note key presses, set hero properties
public function keyDownFunction(event:KeyboardEvent) {
    if (gameMode != "play") return; // don't move until in play mode

    if (event.keyCode == 37) {
        hero.moveLeft = true;
    } else if (event.keyCode == 39) {
        hero.moveRight = true;
    } else if (event.keyCode == 32) {
        if (!hero.inAir) {
            hero.jump = true;
        }
    }
}
```

The keyUpFunction recognizes when the player releases the key and subsequently turns off the moveLeft and moveRight flags:

```
public function keyUpFunction(event:KeyboardEvent) {
     if (event.keyCode == 37) {
          hero.moveLeft = false;
     } else if (event.keyCode == 39) {
          hero.moveRight = false;
     }
}
```

The Main Game Loop

Thanks to the EVENT_FRAME listener, the gameLoop function will be called once per frame. It determines how many milliseconds have passed since the last time it was called.

If the gameMode is "play", it will call a variety of functions. First, it calls moveCharacter with the hero as the object. It also passes in the timeDiff to moveCharacter.

Next, it calls moveEnemies, which basically loops though the enemies and calls moveCharacter for each enemy.

The checkForCollisions function will see whether any enemies collide with the hero, or if the hero gets an item.

Finally, the scrollWithHero will keep the gamelevel movie clip in pace with the hero's position, if needed:

```
public function gameLoop(event:Event) {

     // get time differentce
     if (lastTime == 0) lastTime = getTimer();
     var timeDiff:int = getTimer()-lastTime;
     lastTime += timeDiff;

     // only perform tasks if in play mode
     if (gameMode == "play") {
          moveCharacter(hero,timeDiff);
          moveEnemies(timeDiff);
          checkCollisions();
          scrollWithHero();
     }
}
```

The moveEnemies function checks hitWallRight and hitWallLeft properties of each enemy. These are special properties assigned to a character object when it is processed by moveCharacter. We don't use them with regard to the hero object, but we do for enemies.

When an enemy hits a wall, we will reverse its direction:

```
public function moveEnemies(timeDiff:int) {
    for(var i:int=0;i<enemies.length;i++) {

        // move
        moveCharacter(enemies[i],timeDiff);

        // if hit a wall, turn around
        if (enemies[i].hitWallRight) {
            enemies[i].moveLeft = true;
            enemies[i].moveRight = false;
        } else if (enemies[i].hitWallLeft) {
            enemies[i].moveLeft = false;
            enemies[i].moveRight = true;
        }
    }
}
```

NOTE

It might be desirable for different enemies to have different behaviors. For instance, you could check to see whether the `hero` is to the left or the right of the enemy, and only move in that direction. Or, you could check the distance between the `hero` and the enemy and only move if the `hero` is close.

Character Movement

Now it is time to examine the heart of the game: the `moveCharacter` function. It takes a `character` object, either the `hero` or an `enemy`, and stores it in `char`. It also takes the number of milliseconds that have elapsed and stores that in `timeDiff`:

```
public function moveCharacter(char:Object,timeDiff:Number) {
```

At the start of the game, the `lastTime` variable is initialized, and the resulting time difference will be 0. A value of 0 will not play well with some of the velocity calculations, so we will cut out of the function at this point if `timeDiff` is 0:

```
    if (timeDiff < 1) return;
```

NOTE

If the `timeDiff` is 0, the `verticalChange` will be 0. If the `verticalChange` is 0, the new vertical position and the old vertical position will be the same, which makes it hard to tell whether the character is resting on the ground or hanging in mid-air.

The first thing we need to do is calculate vertical change due to gravity. Gravity is always acting on us, even when we are standing on the ground. We'll calculate what the change is to the character's velocity and vertical position, based on the `gravity` constant and the amount of time that has passed.

To determine the amount of vertical change due to gravity in time based animation, we take the current vertical speed (`char.dy`) and multiply it by the `timeDiff`. This will include the up or down speed that the character currently has.

Then, we add the `timeDiff` times `gravity` to estimate the distance traveled since the last time vertical speed, `dy`, was updated.

Then, we change the vertical speed for future use by adding `gravity*timeDiff`:

```
// assume character pulled down by gravity
var verticalChange:Number = char.dy*timeDiff + timeDiff*gravity;
if (verticalChange > 15.0) verticalChange = 15.0;
char.dy += timeDiff*gravity;
```

NOTE

Notice that the `verticalChange` is limited to 15.0. This is what is known as *terminal velocity*. In real life, this happens when wind resistance counteracts the acceleration due to gravity and the object cannot fall any faster. We add this in here because if the character falls from a high distance, he will accelerate quite a bit, and the effect doesn't look quite right to the eye.

Before we look at left and right movement, we'll make some assumptions about what is going to happen. We'll assume that the animation state will be `"stand"`, and the new direction for the character will be the same as the current direction. We'll also assume that there will be no horizontal change in position:

```
// react to changes from key presses
var horizontalChange = 0;
var newAnimState:String = "stand";
var newDirection:int = char.direction;
```

Then, we'll immediately test that assumption by looking at the `char.moveLeft` and `char.moveRight` properties. These would have been set in the `keyDownFunction` if the player has either the left- or right-arrow key pressed.

If the left key is pressed, the `horizontalChange` is set to negative the `char.walkSpeed*timeDiff`. Also, `newDirection` is set to `-1`. If the right key is pressed, `horizontalChange` is set to positive `char.walkSpeed*timeDiff`, and the `newDirection` is set to `1`. In either case, `newAnimState` is set to `"walk"`:

```
if (char.moveLeft) {
    // walk left
    horizontalChange = -char.walkSpeed*timeDiff;
```

```
        newAnimState = "walk";
        newDirection = -1;
    } else if (char.moveRight) {
        // walk right
        horizontalChange = char.walkSpeed*timeDiff;
        newAnimState = "walk";
        newDirection = 1;
}
```

The next thing we will check for is `char.jump`, which will be set to `true` when the player presses the spacebar. We'll immediately set that to `false` so that the action only occurs once per spacebar press.

Then, we'll change `char.dy` to a negative value of the `char.jumpSpeed` constant. This will give the character an upward push which is the initial force of the jump.

We'll also set the `newAnimState` to `"jump"`. Figure 11.10 shows the hero's jump state.

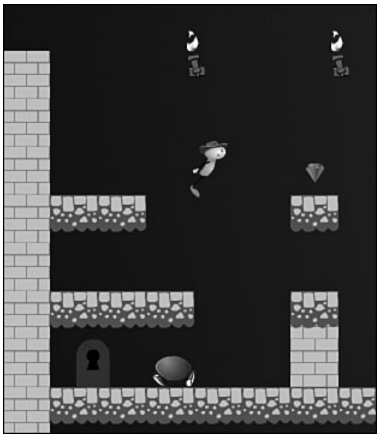


Figure 11.10
Whenever the hero is in the air, it will appear this way.

```
    if (char.jump) {
        // start jump
        char.jump = false;
        char.dy = -char.jumpSpeed;
        verticalChange = -char.jumpSpeed;
        newAnimState = "jump";
    }
```

Now we are about to look at the `fixedObjects` in the scene to check for movement collisions. Before we do that, we'll assume that there is no left or right wall collision and that the character remains in the air:

```
// assume no wall hit, and hanging in air
char.hitWallRight = false;
char.hitWallLeft = false;
char.inAir = true;
```

We'll calculate the new vertical position of the character, based on the current position and the `verticalChange` set earlier:

```
// find new vertical position
var newY:Number = char.mc.y + verticalChange;
```

Now we'll look at each fixed object and see whether any are right under the character's feet. To do this, we'll first look to see whether the character is horizontally aligned with the object. If it is too far to the left or right, we don't have to examine the object further.

Figure 11.11 shows an example of this. Rectangle A shows the character in the current position, and rectangle B shows the character in a future position. You can see that the bottom of the character is on top of the floor in A, and below the floor in B.

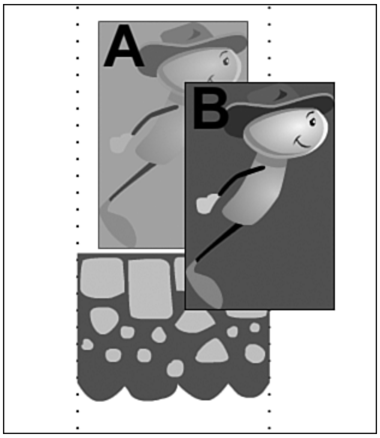


Figure 11.11
In one step, the character would pass through the floor if our code did not stop it.

Next, we'll see whether the character is currently above the object, and whether its `newY` location is below it. This means that the character would normally pass through the object. Remember that the registration point for the characters is at the bottom of their feet, and the registration point for the walls and floors is at the top.

Instead of letting the character pass through the object, we'll stop it right on the object's top surface. The `char.dy` property will be set to `0`, and the `char.inAir` property to `false`:

```
// loop through all fixed objects to see if character has landed
for(var i:int=0;i<fixedObjects.length;i++) {
    if ((char.mc.x+char.width/2 > fixedObjects[i].leftside) &&
            (char.mc.x-char.width/2 < fixedObjects[i].rightside)) {
```

```
            if ((char.mc.y <= fixedObjects[i].topside) &&
                        (newY > fixedObjects[i].topside)) {
                newY = fixedObjects[i].topside;
                char.dy = 0;
                char.inAir = false;
                break;
            }
        }
    }
```

NOTE

While a character is resting on top of a `Floor` or `Wall` piece, this vertical test is being performed with each step, and with each step it will result in the character remaining on top of the floor piece.

Next, we'll perform a similar test with the horizontal position. We'll create a `newX` variable with the new horizontal location, assuming no collisions:

```
// find new horizontal position
var newX:Number = char.mc.x + horizontalChange;
```

Now we'll look at each `Wall` and `Floor` object and see whether any match up vertically. If they do, we'll see whether any are being crossed with the transition from the current position to the new one.

We need to look both to the left and right. If either test is `true`, the `x` position is set to match the wall exactly and `char.hitWallLeft` or `char.hitWallRight` is set to `true`:

```
// loop through all objects to see if character has bumped into a wall
for(i=0;i<fixedObjects.length;i++) {
    if ((newY > fixedObjects[i].topside) &&
            (newY-char.height < fixedObjects[i].bottomside)) {
        if ((char.mc.x-char.width/2 >= fixedObjects[i].rightside) &&
                    (newX-char.width/2 <= fixedObjects[i].rightside)) {
            newX = fixedObjects[i].rightside+char.width/2;
            char.hitWallLeft = true;
            break;
    }
        if ((char.mc.x+char.width/2 <= fixedObjects[i].leftside) &&
                    (newX+char.width/2 >= fixedObjects[i].leftside)) {
            newX = fixedObjects[i].leftside-char.width/2;
            char.hitWallRight = true;
            break;
        }
    }
}
```

Now we know the new position of the character, taking into account horizontal and vertical speed, gravity, and floor and wall collisions. We can set the location of the character:

```
// set position of character
char.mc.x = newX;
char.mc.y = newY;
```

The rest of the function deals with the appearance of the character. We'll check the `char.inAir` value; if it is `true` at this point, we need to set the `newAnimState` to `"jump"`:

```
// set animation state
if (char.inAir) {
     newAnimState = "jump";
}
```

We are done changing the `newAnimState`. This variable started as `"stand"`. Then it changed to `"walk"` if either the left- or right-arrow key was pressed. It could also change to `"jump"` if the player presses the spacebar, or if the character is in the air. Now we'll set the `animstate` to the value of `newAnimState`:

```
char.animstate = newAnimState;
```

Next, we'll use the `animstate` to decide how the character should look. If the character is walking, the `animstep` is increased by a fraction of the `timeDiff`, and a check is made to see whether the `animstep` should loop back around to 0. Then, the frame of the character is set according to the frame specified in `walkAnimation`:

```
// move along walk cycle
if (char.animstate == "walk") {
     char.animstep += timeDiff/60;
     if (char.animstep > char.walkAnimation.length) {
          char.animstep = 0;
     }
     char.mc.gotoAndStop(char.walkAnimation[Math.floor(char.animstep)]);
```

If the character is not walking, we'll set the frame to either `"stand"` or `"jump"` depending on the value of `animstate`:

```
// not walking, show stand or jump state
} else {
     char.mc.gotoAndStop(char.animstate);
}
```

The last thing that `moveCharacter` needs to do is set the orientation of the character. The `direction` property is `-1` for facing left and `1` for facing right. We'll populate it with the `newDirection` that was determined previously. Then, we'll set the `scaleX` property of the character.

NOTE

Setting the `scaleX` of a sprite or movie clip is a simple way to flip the any object. However, if you have shadows or 3D perspective in the object's graphics, you need to draw a separate version to face the other direction; otherwise, the flipped character will not look quite right.

```
        // changed directions
        if (newDirection != char.direction) {
            char.direction = newDirection;
            char.mc.scaleX = char.direction;
        }
    }
```

Scrolling the Game Level

Another function performed every frame is the `scrollWithHero`. This checks the position of the `hero` relative to the stage. The `stagePosition` is calculated by adding the `gamelevel.x` to the `hero.mc.x`. Then, we also get the `rightEdge` and `leftEdge` based on the edges of the screen, minus the `edgeDistance` constant. These are the points at which the screen begins to scroll if needed.

If the `hero` is past the `rightEdge`, the position of the `gamelevel` is moved the same distance to the left. However, if the `gamelevel` is too far to the left, it is restricted from moving so that the right end of the `gamelevel` is at the right side of the stage.

Likewise, if the `hero` is far enough to the left, the `gamelevel` movie clip moves to the right, but only far enough so that the side of `gamelevel` doesn't move to the right of the left side of the screen:

```
    // scroll to the right or left if needed
    public function scrollWithHero() {
        var stagePosition:Number = gamelevel.x+hero.mc.x;
        var rightEdge:Number = stage.stageWidth-edgeDistance;
        var leftEdge:Number = edgeDistance;
        if (stagePosition > rightEdge) {
            gamelevel.x -= (stagePosition-rightEdge);
            if (gamelevel.x < -(gamelevel.width-stage.stageWidth))
                    gamelevel.x = -(gamelevel.width-stage.stageWidth);
        }
        if (stagePosition < leftEdge) {
            gamelevel.x += (leftEdge-stagePosition);
            if (gamelevel.x > 0) gamelevel.x = 0;
        }
    }
```

Checking for Collisions

The checkCollisions function loops through all the enemies and then all the otherObjects. It uses a simple hitTestObject function per object to perform the collision tests.

If the hero and enemy collision occurs while the hero is in the air, and traveling downward, the enemy is destroyed by calling enemyDie. However, if that is not the case, heroDie is called:

```
// check collisions with enemies, items
public function checkCollisions() {

    // enemies
    for(var i:int=enemies.length-1;i>=0;i--) {
        if (hero.mc.hitTestObject(enemies[i].mc)) {

            // is the hero jumping down onto the enemy?
            if (hero.inAir && (hero.dy > 0)) {
                enemyDie(i);
            } else {
                heroDie();
            }
        }
    }
```

If the hero collides with an object in the otherObjects list, getObject is called with the number of the item in the list:

```
    // items
    for(i=otherObjects.length-1;i>=0;i--) {
        if (hero.mc.hitTestObject(otherObjects[i])) {
            getObject(i);
        }
    }
}
```

Enemy and Player Death

When an enemy object is destroyed, it is removed from the gamelevel movie clip, and from the enemies list. That's all it takes for it to disappear.

However, we'll throw in a special effect. By using the PointBurst class from Chapter 8, we can have some text appear at the location where the enemy was removed. In this case, the words "Got Em!" appear. Figure 11.12 shows the screen right after an enemy is destroyed.

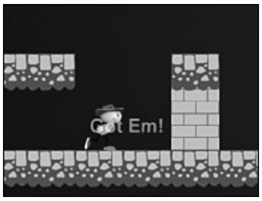


Figure 11.12
The words "Got Em!" appear in the space where the enemy was, and scale up and fade away quickly.

```
// remove enemy
public function enemyDie(enemyNum:int) {
     var pb:PointBurst = new PointBurst(gamelevel,
          "Got Em!",enemies[enemyNum].mc.x,
          enemies[enemyNum].mc.y-20);
     gamelevel.removeChild(enemies[enemyNum].mc);
     enemies.splice(enemyNum,1);
}
```

NOTE
To use the `PointBurst` class, you need to drag a copy of it into the same folder as the **`PlatformGame.fla`** and **`PlatformGame.as`**. You'll also need to add the Arial font to the **PlatformGame.fla** library and set it to Export with ActionScript.

When the player dies as a result of running into an enemy, we'll get the chance to bring up the dialog box that was created earlier.

To create the dialog, we just need to create a new `Dialog` object and assign it to a temporary variable. Then, we set the `x` and `y` position and `addChild` to put it on the stage.

Next, we check the number of `playerLives`. If it is at 0, we set the dialog box text to "Game Over!" and the `gameMode` to `"gameover"`. However, if there are still lives left, we subtract one and set the message to "He Got You!" and the `gameMode` to `"dead"`.

The `gameMode` plays an important part in what happens when the player presses the button inside the dialog box:

```
// enemy got player
public function heroDie() {
     // show dialog box
     var dialog:Dialog = new Dialog();
     dialog.x = 175;
     dialog.y = 100;
     addChild(dialog);

     if (playerLives == 0) {
          gameMode = "gameover";
          dialog.message.text = "Game Over!";
```

```
        } else {
            gameMode = "dead";
            dialog.message.text = "He Got You!";
            playerLives--;
        }

        hero.mc.gotoAndPlay("die");
    }
```

The last thing that the `heroDie` function does is to tell the `hero` movie clip to play from the frame "die". This function begins an animation that shows the player falling down. There is a `stop` command at the end of the `hero` timeline so that the move clip does not loop back around. Figure 11.13 shows the hero dead, and the dialog box displayed.

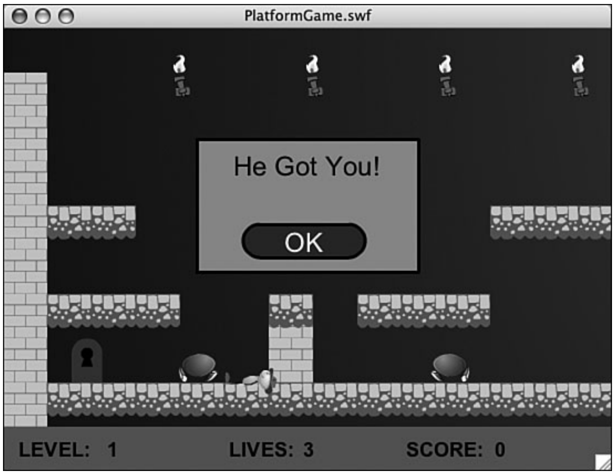


Figure 11.13
The hero is dead, and now the player must press the button to start a new life.

Collecting Points and Objects

When the player collides with an object in the `otherObjects` array, he either get points, an inventory item, or the level will end.

If the object type is `Treasure`, the player gets 100 points. We'll use the `PointBurst` again here to show "100" at the location. Then, we'll remove the object from `gamelevel` and from `otherObjects`. We'll call the `addScore` function to add 100 points and update the score:

```
// player collides with objects
public function getObject(objectNum:int) {

    // award points for treasure
    if (otherObjects[objectNum] is Treasure) {
        var pb:PointBurst = new PointBurst(gamelevel,100,
                otherObjects[objectNum].x,otherObjects[objectNum].y);
        gamelevel.removeChild(otherObjects[objectNum]);
        otherObjects.splice(objectNum,1);
        addScore(100);
```

NOTE

One easy way to have different point values for different `Treasures` is to use the instance name of the `Treasure`. As the game stands, that is not being used by anything else. So, you could set one `Treasure`'s name to `"100"` and another to `"200"`. Then, you could look at `otherObjects[objectNum].name` to assign a point value.

If the object is a `Key`, we'll use `PointBurst` to display the message "Got Key!" We'll add the string `"Key"` to the `playerObjects` array, which acts as an inventory. The object is then removed from `gamelevel` and `otherObjects`:

```
    // got the key, add to inventory
    } else if (otherObjects[objectNum] is Key) {
        pb = new PointBurst(gamelevel,"Got Key!",
                 otherObjects[objectNum].x,otherObjects[objectNum].y);
        playerObjects.push("Key");
        gamelevel.removeChild(otherObjects[objectNum]);
        otherObjects.splice(objectNum,1);
```

Another possibility is that the object is a `Door`. In this case, we'll check the `playerObjects` inventory to see whether `"Key"` is there. If they player has gotten the key, the door will open. We'll do this by telling the `Door` to play starting at frame "open". Then, we'll call `levelComplete`, which displays a dialog box:

```
    // hit the door, end level if hero has the key
    } else if (otherObjects[objectNum] is Door) {
        if (playerObjects.indexOf("Key") == -1) return;
        if (otherObjects[objectNum].currentFrame == 1) {
            otherObjects[objectNum].gotoAndPlay("open");
            levelComplete();
        }
```

The final possibility is that the player has found the `Chest`. This signals the end of the second level, and the end of the player's quest. This movie clip also has an "open" frame, although we'll use `gotoAndStop` because there is no animation there. Then `gameComplete` is called:

```
    // got the chest, game won
    } else if (otherObjects[objectNum] is Chest) {
        otherObjects[objectNum].gotoAndStop("open");
        gameComplete();
    }
}
```

Showing Player Status

Now it is time to look at some utility functions. These are called at various places in the game when needed. This first one adds a number of points to the `gameScore`, and then updates the `scoreDisplay` text field on the stage:

```
// add points to score
public function addScore(numPoints:int) {
    gameScore += numPoints;
    scoreDisplay.text = String(gameScore);
}
```

This next function places the value of `playerLives` into the text field `livesDisplay`:

```
// update player lives
public function showLives() {
    livesDisplay.text = String(playerLives);
}
```

Ending the Levels and the Game

The first level ends when the player gets the key and opens the door. The second level ends when the player finds the treasure chest. In either case, a `Dialog` object is created and placed in the center of the screen.

In the case of opening the door and completing level one, the dialog says "Level Complete!" and the `gameMode` is set to `"done"`:

```
// level over, bring up dialog
public function levelComplete() {
    gameMode = "done";
    var dialog:Dialog = new Dialog();
    dialog.x = 175;
    dialog.y = 100;
    addChild(dialog);
    dialog.message.text = "Level Complete!";
}
```

In the case of the end of level two, when the player finds the chest, the message will read "You Got the Treasure!" and `gameMode` will be `"gameover"`:

```
// game over, bring up dialog
public function gameComplete() {
    gameMode = "gameover";
    var dialog:Dialog = new Dialog();
    dialog.x = 175;
    dialog.y = 100;
    addChild(dialog);
    dialog.message.text = "You Got the Treasure!";
}
```

The Game Dialog Box

The dialog box appears when the player has died, completed a level, or completed the game. When the player presses the button in the dialog box, it will call the `clickDialogButton` function in the main class. Here is the code from inside the `Dialog` object:

```
okButton.addEventListener(MouseEvent.CLICK,MovieClip(parent).clickDialogButton);
```

The first thing the `clickDialogButton` function does is to remove the dialog itself:

```
// dialog button clicked
public function clickDialogButton(event:MouseEvent) {
    removeChild(MovieClip(event.currentTarget.parent));
```

The next thing it does depends on the value of `gameMode`. If the player is dead, the display of lives is updated, the `hero` is put back to the position it was in when the level started, and the `gameMode` is set back to `"play"` so that play can continue:

```
    // new life, restart, or go to next level
    if (gameMode == "dead") {
        // reset hero
        showLives();
        hero.mc.x = hero.startx;
        hero.mc.y = hero.starty;
        gameMode = "play";
```

If the `gameMode` is `"gameover"`, which will happen if the player dies for the last time, or if the player finds the treasure chest, the `cleanUp` function is called to remove the `gamelevel` movie clip, and the movie is sent back to the start:

```
    } else if (gameMode == "gameover") {
        cleanUp();
        gotoAndStop("start");
```

The other option is that the `gameMode` is `"done"`. This means it is time to go to the next level. The `cleanUp` function is called again, and then the movie is sent to the next frame, where a new version of `gamelevel` awaits:

```
    } else if (gameMode == "done") {
        cleanUp();
        nextFrame();
    }
```

One last thing that must be done is to return the keyboard focus to the stage. The stage loses focus when the button is clicked. We want to make sure that arrow keys and the spacebar are routed back to the stage again:

```
    // give stage back the keyboard focus
    stage.focus = stage;
}
```

The `cleanUp` function that the `clickDialogButton` function calls removes the `gamelevel` and also removes the listeners that were applied to the stage, as well as the `ENTER_FRAME` listener. These will be re-created in `startLevel` if the player is to go on to the next level:

```
// clean up game
public function cleanUp() {
     removeChild(gamelevel);
     this.removeEventListener(Event.ENTER_FRAME,gameLoop);
     stage.removeEventListener(KeyboardEvent.KEY_DOWN,keyDownFunction);
     stage.removeEventListener(KeyboardEvent.KEY_UP,keyUpFunction);
}
```

Modifying the Game

For this game to become something real and challenging, more levels need to be added, with more elements. You can add as many levels as you want.

Right now, the first level ends when the key is in the inventory and the door is found. You'll probably want to alter the code to add other options, like a door that doesn't need a key, or a door that needs more than one.

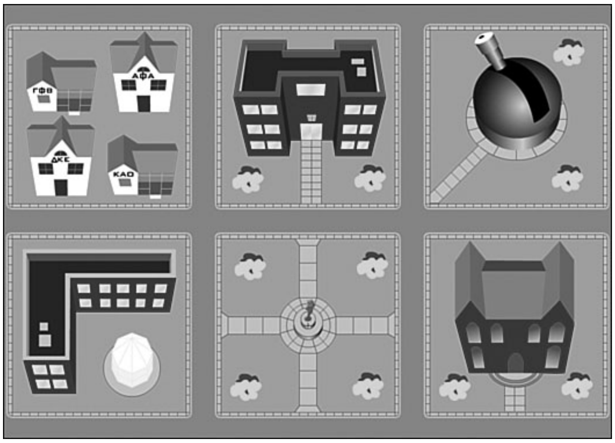
Another feature that would make the game more interesting would be more ways to die. Right now, you can only die if an enemy touches you while you are not jumping. And, it is pretty easy to kill them off.

But what if there were blocks or objects that could kill you, too. For instance, there could be pits of animated lava. You'd need to jump over these pits just perfectly to make it to a goal.

There could also be more animated hazards, like spears that shoot out of the walls. These would be just like enemies, but when they collide with the opposite wall, they will "die." Then, you could set a `Timer` to create a new spear at regular intervals.

The possibilities are literally endless. But although it is easy to make a creative and fun platform game, it is also easy to make a bad one. So, think carefully about your game design, and test and tweak your design along the way.

This page intentionally left blank



12

Game Worlds: Driving and Exploration Game

- Creating a Top-Down Driving Game
- Building a Flash Racing Game

In the preceding chapter, you saw how it was possible to create a small world inside an ActionScript game. This type of platform game creates a side view that is usually used for indoor adventures and quests.

Another type of game world can be done with a top-down view. The player is then moving around on a map. This can fit almost any scenario and theme. Recently, however, I have noticed quite a few top-down games where the player drives a vehicle around a town or other outdoor location.

In this chapter, we look at a top-down driving game and a straightforward racing game. Both types of games have some things in common.

Creating a Top-Down Driving Game

Let's create a simple top-down driving game. This game features a detailed map, a car, objects to collect, and a complex game logic involving a place to deposit the objects collected.

SOURCE FILES
http://flashgameu.com
A3GPU12_TopDownGame.zip

Creating a Top-Down World

Our example game for this chapter features a college campus. There will be a three-block by three-block area, and various buildings coloring in the spaces between the streets. Figure 12.1 shows the campus.

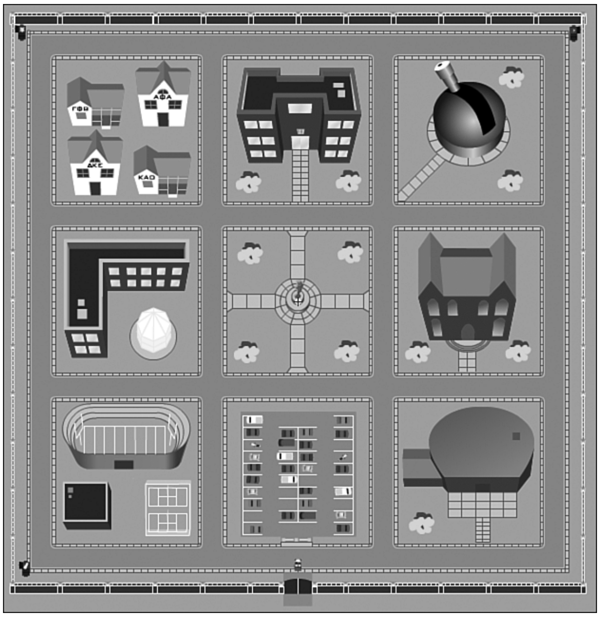


Figure 12.1
The entire game world is about 2,400 pixels wide and 2,400 high.

If you look closely at the gate near the bottom of the map, you'll see a small car. This is the player's car, and he or she will be able to "drive" around the map using it.

Because the map is so big, the player won't be able to see more than a small section of it at a time. The map is 2,400 pixels square, and the screen is 550x400.

As the player drives, the map will reposition itself with the location of the car at the exact center of the stage.

Figure 12.2 shows the screen when the player starts. You can see the gate at the bottom, and a bit of the parking lot above it. At the bottom is a semitransparent strip with score elements in it.

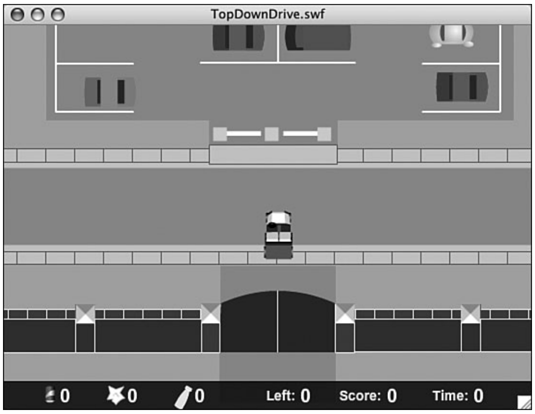


Figure 12.2
Only a small 550x400 area of the map can be seen at any given time.

The map is all located in a single movie clip named `GameMap`. Inside it, the nine building groups each have their own movie clip, for organizational purposes. The streets are made up of straight pieces and three different types of corners. The outer fence is made up of a few different pieces, too.

All of these graphic elements are just for decoration. They aren't actually important to the game code. This is great news for artists because it means they can have free reign on creating an artistic backdrop for the game.

The car will be able to move around the screen anywhere, with only a few simple restrictions.

First, the car will be restricted to the area inside the fence. This will be defined by minimum and maximum `x` and `y` values.

Second, the car will be restricted from entering the area of certain other movies clips. We'll call these `Blocks`. If the car collides with one of these `Blocks`, it will stop at the edge.

NOTE

The use of the term *block* has three meanings. Most important, it will block the car from entering an area. But, they also represent city blocks in this map. In addition, they are rectangular in shape.

The nine `Blocks` will be placed over the nine city blocks in the map. Figure 12.3 shows the locations of these with thick borders.

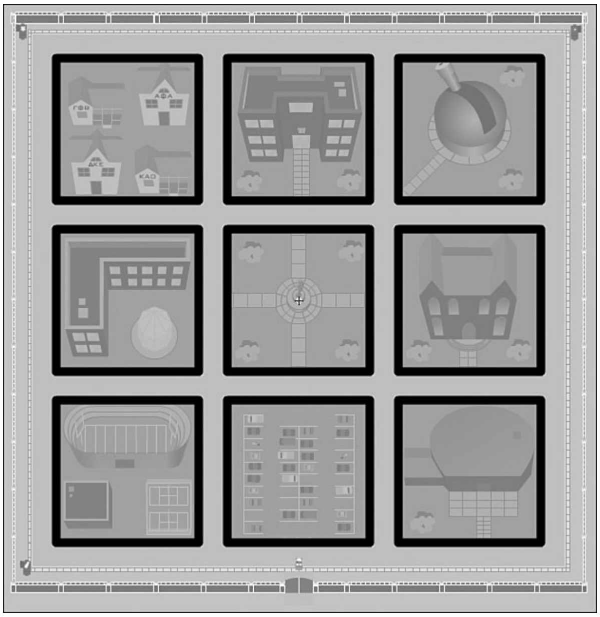


Figure 12.3
The nine `Block` *movie clips are shown with thick outlines.*

The object of the game is to collect trash around campus and deposit it in recycling bins. There will be three recycling dumpsters placed in three of the corners of the campus.

There will be three different types of trash, one for each dumpster: cans, paper, and bottles.

Instead of placing the trash items in the map by hand, we'll have our code do it. It will place 100 different pieces of trash randomly on campus. We need to make sure they are not on `Blocks`; otherwise, the car will not be able to get to them.

The challenge is to collect all the trash and deposit each type into its own bin. However, the car can only hold ten different pieces of trash at a time. Before players can pick up any more, they must visit a dumpster and deposit some trash.

So, the game gets challenging as players must decide which pieces of trash to pick up based on which bin they are heading for.

Game Design

It is worth taking a look at all of the game inputs, objects, and mechanisms before we start programming. This will help clarify what we need to do.

Car Control

The car will be controlled by the arrow keys. In fact, only three of the four arrow keys will be needed. Figure 12.4 shows the `car` movie clip.

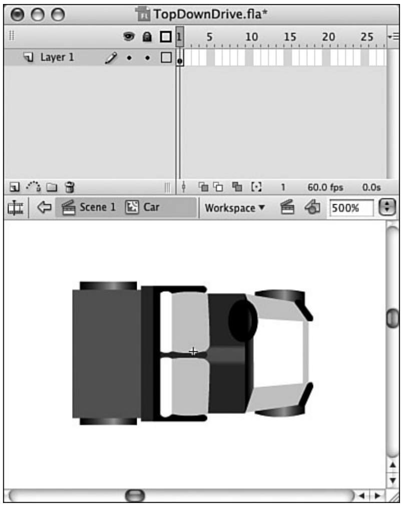


Figure 12.4
The car movie clip is pointed to the right, so 0 `rotation` *matches the direction that* `Math.cos` *and* `Math.sin` *represent.*

We're not creating a simulation here, so things such as acceleration, braking, and reversing can be ignored so long as the player doesn't need them. In this case, being able to steer left and right and move forward is fine for getting around.

We'll use the left and right arrow keys to directly change the `rotation` property of the car. Then, we'll use the `Math.cos` and `Math.sin` values of the `rotation` to determine forward movement. This is similar to how we used arrow keys and trigonometry in the space rocks game from Chapter 7, "Direction and Movement: Space Rocks."

Car Boundaries

The car is restricted to the streets. Or, to be more precise, the car cannot leave the map, and it cannot run over any of the `Block` movie clips. The `Block` movie clip can be seen in Figure 12.5.

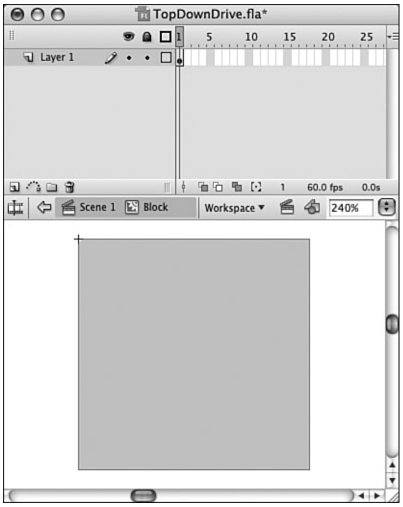


Figure 12.5
The `Block` *is never seen except by us as we design the level. A thin red border and a semitransparent fill will help us place them.*

To do this, we'll compare the rectangle of the car to the `Block` rectangles. We'll get a list of them when the game first starts. If the car's rectangle and any one of the `Blocks` intersect, we'll push the car back to the point where it is just outside of the `Block`.

This is similar to how the paddle ball game worked in Chapter 5, "Game Animation: Shooting and Bouncing Games." However, instead of bouncing the car off the `Block`, we'll just set it perfectly so it is just outside of the `Block`.

Trash

The trash is actually a single `TrashObject` movie clip with three frames. We'll place them randomly on the map, making sure that none are placed on the `Blocks`.

When one is placed, it will be randomly set to frame 1, 2, or 3, representing one of the three types of trash: cans, paper, or bottles. Figure 12.6 shows the `TrashObject` movie clip.

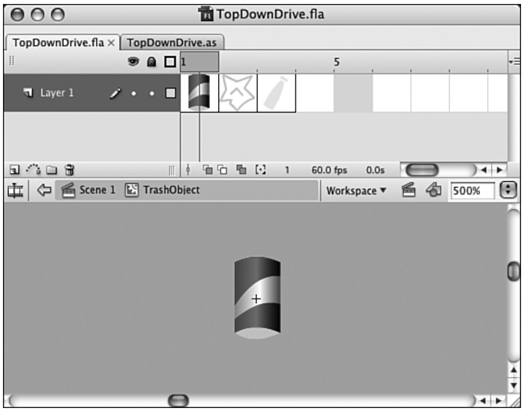


Figure 12.6
The `TrashObject` *movie clip has three different frames, each with a different type of trash on them.*

As the car moves around, we'll look for the distance between each `TrashObject` and the car to be close enough so the car will pick it up.

We'll remove these from the screen and keep track of how much of each type of trash the player has. We'll limit that to ten items at a time and indicate to the player when they are full.

Then, when the player gets close to a dumpster, we'll zero out one of the kinds of items in the player's collection. A smart player will fill up on only one type of trash, and then dump all ten of those items at the proper dumpster.

Game Score and Clock

The score indicators, shown at the bottom of Figure 12.7, are more important in this game than in others we have made so far. The player must pay careful attention to them.

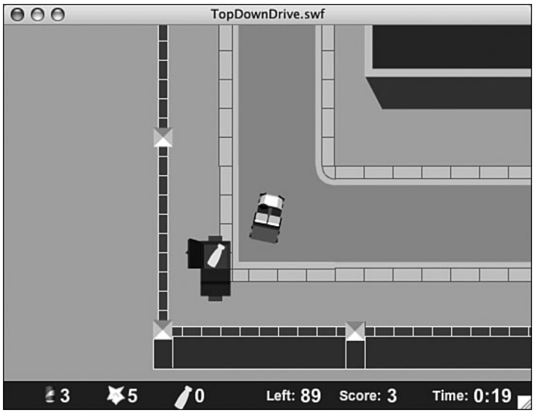


Figure 12.7
The score indicators are at the bottom of the screen, with a semitransparent box under them.

The first three indicators are the number of trash items the player has. Because players can only have ten items before going to a dumpster, they will want to get mostly one type of item. And, they will want to pay attention to when they are getting close to full.

We'll have all three numbers turn red when the car is full of trash. We'll also use sound to indicate this. There will be a pickup sound when the player drives near a piece of trash. If the car is full, however, they will get a different sound instead, and the trash will remain on the street.

The next two indicators show the number of trash items left to find, the number found, and the time. The time is the key value here. Players will always find all 100 pieces of trash, unless they quit early. The time will be their score. Playing the game well will mean finishing in less time.

The Class Definition

The code for this game is fairly simple considering all that the game does. The game starts by examining the world created in the Flash movie, and then checks every frame for player changes and movement.

The package starts off by importing a wide range of class libraries. We need the usual suspects, plus `flash.geom.*` for use of the `Point` and `Rectangle` objects and `flash.media.Sound` and `flash.media.SoundChannel` for sound effects:

```
package {
     import flash.display.*;
     import flash.events.*;
     import flash.text.*;
     import flash.geom.*;
     import flash.utils.getTimer;
     import flash.media.Sound;
     import flash.media.SoundChannel;
```

The game has quite a few constants. The `speed` and `turnSpeed` control how the car reacts to the arrow keys. The `carSize` determines the boundary rectangle of the car from its center point:

```
public class TopDownDrive extends MovieClip {

     // constants
     static const speed:Number = .3;
     static const turnSpeed:Number = .2;
     static const carSize:Number = 50;
```

The `mapRect` constant defines the boundaries of the map. This is approximately the location of the fence surrounding the campus:

```
     static const mapRect:Rectangle = new Rectangle(-1150,-1150,2300,2300);
```

The `numTrashObjects` constant is the number of pieces of trash created at the start of the game. We also have the `maxCarry` to set the number of pieces of trash that the player can have in the car before they need to empty out at a dumpster:

```
     static const numTrashObjects:uint = 100;
     static const maxCarry:uint = 10;
```

The next two constants set the distance for trash and trashcan collisions. You may need to adjust this number if you move the trashcans further off the road, or change the `carSize` constant:

```
     static const pickupDistance:Number = 30;
     static const dropDistance:Number = 40;
```

NOTE

You don't want to make `pickUpDistance` too large because it will be important for players to sneak the car past some pieces of trash if they are only collecting trash of one type.

The variables can be divided into three groups. The first group is a series arrays that keeps track of the game objects.

The `blocks` array will contain all of the `Block` objects that prevent the car from leaving the road. The `trashObjects` is a list of all the trash items spread randomly around the map. The `trashcans` array contains the three trashcans that are the drop-off points for the trash:

```
// game objects
private var blocks:Array;
private var trashObjects:Array;
private var trashcans:Array;
```

The next set of variables all deal with the game state. We start with the usual set of arrow-key Boolean variables:

```
// game variables
private var arrowLeft, arrowRight, arrowUp, arrowDown:Boolean;
```

Next, we've got two time values. The first, `lastTime` will be used to determine the length of time since the last animation step. The `gameStartTime` will be used to determine how long the player has been playing:

```
private var lastTime:int;
private var gameStartTime:int;
```

The `onboard` array is a list with one item for each trashcan—so three items. They will all start at 0, and contain the number of each kind of trash that the player has in the car:

```
private var onboard:Array;
```

The `totalTrashObjects` variable will contain the sum of all three numbers in `onboard`. We'll use it for quick and easy reference when deciding whether there is enough room in the car for more trash:

```
private var totalTrashObjects:int;
```

The `score` is simply the number of trash objects that have been picked up and delivered to trashcans:

```
private var score:int;
```

The `lastObject` variable is used to determine when to play the "can't get more trash because the car is full" sound. When players have ten items already collected, and they collide with a piece of trash, we'll play a negative sound, as opposed to the positive sound they get when they have room for the trash.

Because the trash is not removed from the map, chances are that they will collide with the piece again immediately, and continue to do so until the car moves far enough away from the trash.

So, we'll record a reference to the `Trash` object in `lastObject` and save it for later reference. This way we know that a negative sound already played for this object and not to play it again and again while the car is still near it:

```
private var lastObject:Object;
```

The last variables are references to the four sounds stored in the movie's library. All these sounds have been set with linkage properties so that they exist as classes available for our ActionScript to access:

```
// sounds
var theHornSound:HornSound = new HornSound();
var theGotOneSound:GotOneSound = new GotOneSound();
var theFullSound:FullSound = new FullSound();
var theDumpSound:DumpSou nd = new DumpSound();
```

The `Constructor` Function

When the movie reaches frame 2, it will call `startTopDownDrive` to begin the game.

This function immediately calls `findBlocks` and `placeTrash` to set up the map. We'll look at those functions soon.

```
public function startTopDownDrive() {

    // get blocks
    findBlocks();

    // place trash items
    placeTrash();
```

Because there are only three trashcans and they have been specifically named in the `gamesprite`, we'll place them in the `trashcans` array in one simple line of code.

NOTE

The `gamesprite` is the instance on the stage of the `GameMap` library element. In the library, it is actually a `MovieClip`. Because it is only a single frame, however, we'll call it `gamesprite`.

```
    // set trash cans
    trashcans = new Array(gamesprite.Trashcan1,
        gamesprite.Trashcan2, gamesprite.Trashcan3);
```

Because the `Trash` objects are created by our code, and the `car` exists in the `gamesprite` before our code runs, the trash will be on top of the car. This will be apparent after the car is full and the player is racing past other pieces of trash. You'll see the trash float over the car unless we do something about it. By calling `setchildIndex` with `gamesprite.numChildren-1`, we'll be placing the `car` back on top of everything else in the game:

```
// make sure car is on top
gamesprite.setChildIndex(gamesprite.car,gamesprite.numChildren-1);
```

NOTE

Alternatively, we could have created an empty movie clip in the `GameMap` movie clip to hold all of the trash items. Then, we could have placed it in a timeline layer just below the car, but above the street. This would be important if we want to have some items, such as a bridge, remain on top of both the car and the trash.

We need three listeners, one for the `ENTER_FRAME` event, which will run the entire game. The other two are for the key presses:

```
// add listeners
this.addEventListener(Event.ENTER_FRAME,gameLoop);
stage.addEventListener(KeyboardEvent.KEY_DOWN,keyDownFunction);
stage.addEventListener(KeyboardEvent.KEY_UP,keyUpFunction);
```

We set up the game state next. The `gameStartTime` is set to the current time. The `onboard` array is set to all zeros, as well as the `totalTrashObjects` and `score`:

```
// set up game variables
gameStartTime = getTimer();
onboard = new Array(0,0,0);
totalTrashObjects = 0;
score = 0;
```

We'll call two utility functions right away to get the game going. The `centerMap` function is what places the `gamesprite` so that the `car` is at the center of the screen. If we don't call that now, we'll get a flash of how the `gamesprite` appears in the raw timeline before the first `ENTER_FRAME`.

A similar idea is behind calling `showScore` here, so all the score indicators are set to their original values before the player can see them:

```
centerMap();
showScore();
```

Finally, we'll end by playing a sound using the utility function `playSound`. I've included a simple horn honk to signal the player that they game has begun:

```
    playSound(theHornSound);
}
```

Finding the Blocks

To find all `Block` objects in the `gamesprite`, we need to loop through all the children of `gamesprite` and see which ones are `Block` types by using the `is` operator.

If they are, we'll add them to the `blocks` array. We'll also set the `visible` property of each of the `Blocks` to `false` so that they won't appear to the player. This way we can clearly see them while developing the movie, but don't need to remember to hide them or set them to a transparent color before finishing the game:

```
// find all Block objects
public function findBlocks() {
     blocks = new Array();
     for(var i=0;i<gamesprite.numChildren;i++) {
          var mc = gamesprite.getChildAt(i);
          if (mc is Block) {
               // add to array and make invisible
               blocks.push(mc);
               mc.visible = false;
          }
     }
}
```

Placing the Trash

To place 100 random pieces of trash, we need to loop 100 times, placing 1 piece of trash each time:

```
// create random Trash objects
public function placeTrash() {
     trashObjects = new Array();
     for(var i:int=0;i<numTrashObjects;i++) {
```

For each placement, we start a second loop. Then, we try different values for the `x` and `y` position of the trash:

```
          // loop forever
          while (true) {

               // random location
               var x:Number = Math.floor(Math.random()*mapRect.width)+mapRect.x;
               var y:Number = Math.floor(Math.random()*mapRect.height)+mapRect.y;
```

After we have a location, we check it against all of the `Blocks`. If the location is on a `Block`, we note it by setting the `isOnBlock` local variable to `true`:

```
               // check all blocks to see if it is over any
               var isOnBlock:Boolean = false;
               for(var j:int=0;j<blocks.length;j++) {
```

```
            if (blocks[j].hitTestPoint(x+gamesprite.x,y+gamesprite.y)) {
                isOnBlock = true;
                break;
            }
        }
```

If the location doesn't intersect with any `Blocks`, we go ahead and create the new `TrashObject` object. Then, we set its location. We also need to choose a random type for this piece, by setting the movie clip to frame 1, 2, or 3. Figure 12.8 shows the beginning of a game where three `TrashObject` movie clips have been placed near the starting point of the `car`.

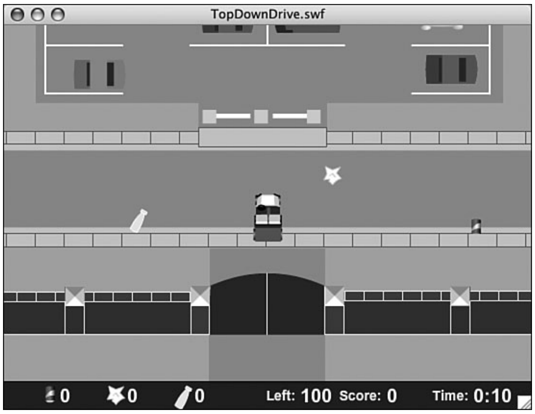


Figure 12.8
Three `TrashObject` *movie clips have been randomly placed near the car at the start of the game.*

NOTE

The `TrashObject` movie clip has three frames, each with a different graphic. These are actually movie clips themselves. Their use in `TrashObject` doesn't need them to be separate movie clips, but we want to be able to use the same graphics for the trashcans to indicate which trashcan can take which type of trash. This way, we only have one version of each graphic in the library.

We'll add this piece of trash to `trashObjects` and then `break`.

This final `break` will exit the `while` loop and move on to placing the next piece of trash. However, if the `isOnBlock` is `true`, we'll continue with the `while` loop by choosing another location to test:

```
        // not over any, so use location
        if (!isOnBlock) {
            var newObject:TrashObject = new TrashObject();
            newObject.x = x;
            newObject.y = y;
            newObject.gotoAndStop(Math.floor(Math.random()*3)+1);
```

```
                    gamesprite.addChild(newObject);
                    trashObjects.push(newObject);
                    break;
                }
            }
        }
    }
```

NOTE

When testing out a placement function such as `placeTrash`, it is useful to try it with the number of objects set very high. For instance, I tested `placeTrash` with a `numTrashObjects` set to 10,000. This littered trash all over the road, but I could see clearly that the trash was only on the road and not in places where I didn't want it.

Keyboard Input

The game includes a set of keyboard input functions similar to the ones we have used in several games up to this point. Four Boolean values are set according to whether the four arrow keys are triggered on the keyboard.

The functions even recognize the down arrow, although this version of the game doesn't use it:

```
// note key presses, set properties
public function keyDownFunction(event:KeyboardEvent) {
    if (event.keyCode == 37) {
        arrowLeft = true;
    } else if (event.keyCode == 39) {
        arrowRight = true;
    } else if (event.keyCode == 38) {
        arrowUp = true;
    } else if (event.keyCode == 40) {
        arrowDown = true;
    }
}

public function keyUpFunction(event:KeyboardEvent) {
    if (event.keyCode == 37) {
        arrowLeft = false;
    } else if (event.keyCode == 39) {
        arrowRight = false;
    } else if (event.keyCode == 38) {
        arrowUp = false;
    } else if (event.keyCode == 40) {
        arrowDown = false;
    }
}
```

The Game Loop

The `gameLoop` function will handle car movement. There are actually no other moving objects in the game. The player moves the car, and everything else remains static inside the `gamesprite`.

This is a time-based animation game, so we'll calculate the time that has passed since the last frame, and move things according to this time value:

```
public function gameLoop(event:Event) {

     // calculate time passed
     if (lastTime == 0) lastTime = getTimer();
     var timeDiff:int = getTimer()-lastTime;
     lastTime += timeDiff;
```

We'll check the left and right arrow keys and call `rotateCar` to handle steering. We'll pass in the `timeDiff` and the direction of the turn:

```
     // rotate left or right
     if (arrowLeft) {
          rotateCar(timeDiff,"left");
     }
     if (arrowRight) {
          rotateCar(timeDiff,"right");
     }
```

If the up arrow is pressed, we'll call `moveCar` with the `timeDiff`. Then, we'll call `centerMap` to make sure the `gamesprite` is positioned correctly with the new location of the car.

The `checkCollisions` function checks to see whether the player has grabbed any trash or has gotten close to a trashcan:

```
     // move car
     if (arrowUp) {
          moveCar(timeDiff);
          centerMap();
          checkCollisions();
     }
```

Remember that the time is the real score in this game. The player is racing the clock. So, we need to update the time for the player to know how he or she is doing:

```
     // update time and check for end of game
     showTime();
}
```

Let's take a look right away at the `centerMap` function, because it is so simple. All it needs to do is to set the location of the `gamesprite` to negative versions of the location of the `car` inside the `gamesprite`. For instance, if the `car` is at location 1000,600 in `gamesprite`, setting the location of the `gamesprite` to –1000,–600 means that the `car` will be at location 0,0 on the stage.

We don't want the `car` at 0,0, which is the upper-left corner. We want it in the center of the stage. So, we add 275,200 to center it.

NOTE

If you want to change the size of the visible area of the stage, say to 640x480, you would also want to change the values here to match the middle of the stage area. So a 640x480 stage would mean 320 and 240 as the `x` and `y` adjustments to place the `car` at the middle of the screen.

```
public function centerMap() {
    gamesprite.x = -gamesprite.car.x + 275;
    gamesprite.y = -gamesprite.car.y + 200;
}
```

Moving the Car

Steering the car is very unrealistic in this game: The car is rotated around its center by a few degrees each frame. In fact, the car can turn without moving forward. Try that in your Toyota.

If you play, however, you'll hardly notice. The rotation is time based, so it is the product of the `timeDiff` and the `turnSpeed` constant. The car should turn at the same rate no matter what the frame rate of the movie:

```
public function rotateCar(timeDiff:Number, direction:String) {
    if (direction == "left") {
        gamesprite.car.rotation -= turnSpeed*timeDiff;
    } else if (direction == "right") {
        gamesprite.car.rotation += turnSpeed*timeDiff;
    }
}
```

Moving the car forward is pretty simple, too, or it would be, if not for the need to detect and deal with collisions between the `Blocks` and the edges of the map.

We'll simplify the collision detection by using simple `Rectangle` objects and the `intersects` function. So, the first thing we need is the `Rectangle` of the `car`.

The `car` is already a rectangular shape, because the `car` rotates, using the movie clip's exact `Rectangle` is a problem. Instead, we'll use a made-up `Rectangle` based on the center of the `car` and the `carSize`. This square area will be a good enough approximation of the area of the `car` that the player won't notice.

NOTE

Keeping the car graphic to a relatively square size, where it is about as long as it is wide, is important to maintaining the illusion of accurate collisions. Having a car that was much longer than wide would require us to base our collision distance depending on the rotation of the car relative to the edge it may be colliding with. And, that would be much more complex.

```
// move car forward
public function moveCar(timeDiff:Number) {
     // calculate current car area
     var carRect = new Rectangle(gamesprite.car.x-carSize/2,
          gamesprite.car.y-carSize/2, carSize, carSize);
```

So, now we have the car's present location in carRect. To calculate the new location of the car, we convert the rotation of the car to radians, feed those numbers to Math.cos and Math.sin, and then multiply those values by the speed and timeDiff. This gives us time-based movement using the speed constant. Then, newCarRect holds the new location of the car:

```
     // calculate new car area
     var newCarRect = carRect.clone();
     var carAngle:Number = (gamesprite.car.rotation/360)*(2.0*Math.PI);
     var dx:Number = Math.cos(carAngle);
     var dy:Number = Math.sin(carAngle);
     newCarRect.x += dx*speed*timeDiff;
     newCarRect.y += dy*speed*timeDiff;
```

We also need the x and y location that matches the new Rectangle. We'll add the same values to x and y to get this new location:

```
     // calculate new location
     var newX:Number = gamesprite.car.x + dx*speed*timeDiff;
     var newY:Number = gamesprite.car.y + dy*speed*timeDiff;
```

Now it is time to loop through the blocks and see whether the new location intersects with any of them:

```
     // loop through blocks and check collisions
     for(var i:int=0;i<blocks.length;i++) {

          // get block rectangle, see if there is a collision
          var blockRect:Rectangle = blocks[i].getRect(gamesprite);
          if (blockRect.intersects(newCarRect)) {
```

If there is a collision, we'll look at both the horizontal and vertical aspects of the collision separately.

If the `car` has passed the left side of a `Block`, we'll push the `car` back to the edge of that `Block`. The same idea is used for the right side of the `Block`. We don't need to bother to adjust the `Rectangle`, just the `newX` and `newY` position values. It is these that will be used to set the new location of the `car`:

```
            // horizontal push-back
            if (carRect.right <= blockRect.left) {
                newX += blockRect.left - newCarRect.right;
            } else if (carRect.left >= blockRect.right) {
                newX += blockRect.right - newCarRect.left;
            }
```

Here is the code that handles the top and bottom sides of the colliding `Block`:

```
            // vertical push-back
            if (carRect.top >= blockRect.bottom) {
                newY += blockRect.bottom-newCarRect.top;
            } else if (carRect.bottom <= blockRect.top) {
                newY += blockRect.top - newCarRect.bottom;
            }
        }

    }
```

After all the `blocks` have been examined for possible collisions, we need to look at the map boundaries. This is the opposite of the `blocks`, because we want to keep the `car` *inside* the boundary `Rectangle`, rather than *outside* of it.

So, we'll examine each of the four sides and push back the `newX` or `newY` values to prevent the `car` from escaping the map:

```
    // check for collisions with sidees
    if ((newCarRect.right > mapRect.right) && (carRect.right <= mapRect.right)) {
        newX += mapRect.right - newCarRect.right;
    }
    if ((newCarRect.left < mapRect.left) && (carRect.left >= mapRect.left)) {
        newX += mapRect.left - newCarRect.left;
    }

    if ((newCarRect.top < mapRect.top) && (carRect.top >= mapRect.top)) {
        newY += mapRect.top-newCarRect.top;
    }
    if ((newCarRect.bottom > mapRect.bottom) && (carRect.bottom <= mapRect.bottom)) {
        newY += mapRect.bottom - newCarRect.bottom;
    }
```

Now that the `car` is safely inside the map, and outside of any `Block`, we can set the new location of the `car`:

```
     // set new car location
     gamesprite.car.x = newX;
     gamesprite.car.y = newY;
}
```

Checking for Trash and Trashcan Collisions

The `checkCollisions` function needs to look for two different types of collisions. It starts by looking at all the `trashObjects`. It will use the `Point.distance` function to see whether the location of the `car` and the location of the `TrashObject` are closer than the `pickupDistance` constant:

```
public function checkCollisions() {


     // loop through trash cans
     for(var i:int=trashObjects.length-1;i>=0;i--) {

          // see if close enough to get trash objects
          if (Point.distance(new Point(gamesprite.car.x,gamesprite.car.y),
            new Point(trashObjects[i].x, trashObjects[i].y)) < pickupDistance) {
```

If an item is close enough, we'll check `totalTrashObjects` against the `maxCarry` constant. If there is room, the item is picked up by setting the right slot in `onboard` according to the `currentFrame-1` of the `TrashObject` movie clip. Then, it is removed from `gamesprite` as well as the `trashObjects` array. We need to update the score and play the `GotOneSound`:

```
               // see if there is room
               if (totalTrashObjects < maxCarry) {
                    // get trash object
                    onboard[trashObjects[i].currentFrame-1]++;
                    gamesprite.removeChild(trashObjects[i]);
                    trashObjects.splice(i,1);
                    showScore();
                    playSound(theGotOneSound);
```

NOTE

One aspect of our code that can be confusing is the way in which trash item types are referenced. As frames in the `TrashObject` movie clip, they are frames 1, 2, and 3. But, arrays are 0 based; so in the `onboard` array, we store trash types 1, 2, and 3 in array locations 0, 1, and 2. The trashcans will be named `Trashcan1`, `Trashcan2`, and `Trashcan3` and will correspond to the frame numbers, not the array slots. As long as you keep this in mind, you'll be able to avoid any problems in modifying the code. Having 0-based arrays, but frame numbers that start at 1, is a constant problem for ActionScript developers.

On the other hand, if the player has hit an item, but there is no more room, we'll play another sound. We'll play the sound only if the item is not the `lastObject`. This will prevent the sound from playing over and over as the player moves across an object. It will play just once per object hit:

```
                } else if (trashObjects[i] != lastObject) {
                    playSound(theFullSound);
                    lastObject = trashObjects[i];
                }
            }
        }
```

The next set of collisions look at the three trashcans. We'll use `Point.distance` here, too. After a collision is detected, we'll remove any of that type of trash from the `onboard` array. We'll update the score and play a sound to acknowledge the player's achievement:

```
        // drop off trash if close to trashcan
        for(i=0;i<trashcans.length;i++) {

            // see if close enough
            if (Point.distance(new Point(gamesprite.car.x,gamesprite.car.y),
              new Point(trashcans[i].x, trashcans[i].y)) < dropDistance) {

                // see if player has some of that type of trash
                if (onboard[i] > 0) {

                    // drop off
                    score += onboard[i];
                    onboard[i] = 0;
                    showScore();
                    playSound(theDumpSound);
```

If the `score` has risen to the point of the `numTrashObjects` constant, the last piece of trash has been deposited, and the game is over:

```
                    // see if all trash has been dropped off
                    if (score >= numTrashObjects) {
                        endGame();
                        break;
                    }
                }
            }
        }
    }
```

The Clock

Updating the clock is pretty simple, and very similar to what we did in the matching game in Chapter 3, “Basic Game Framework: A Matching Game.” We’ll subtract the current time from the start time to get the number of milliseconds. Then, we’ll use the utility function `clockTime` to convert that to a time format:

```
// update the time shown
public function showTime() {
     var gameTime:int = getTimer()-gameStartTime;
     timeDisplay.text = clockTime(gameTime);
}
```

The `clockTime` function computes the number of seconds and minutes, and then formats it with leading zeros if needed:

```
// convert to time format
public function clockTime(ms:int):String {
     var seconds:int = Math.floor(ms/1000);
     var minutes:int = Math.floor(seconds/60);
     seconds -= minutes*60;
     var timeString:String = minutes+":"+String(seconds+100).substr(1,2);
     return timeString;
}
```

The Score Indicators

Showing the score in this game is much more complex than just showing a single number. We’ll be showing the three numbers stored in `onboard`. At the same time we’ll be adding these numbers up for `totalTrashObjects`, which will be used elsewhere in the game to determine whether there is more room in the car:

```
// update the score text elements
public function showScore() {

     // set each trash number, add up total
     totalTrashObjects = 0;
     for(var i:int=0;i<3;i++) {
          this["onboard"+(i+1)].text = String(onboard[i]);
          totalTrashObjects += onboard[i];
     }
```

We’ll also use `totalTrashObjects` right now to color all three numbers red or white depending on whether the car is full. This gives us a natural indicator for the players to see whether they have maxed out the car’s capacity and need to go to a trashcan:

```
     // set color of all three based on whether full
     for(i=0;i<3;i++) {
          if (totalTrashObjects >= 10) {
                this["onboard"+(i+1)].textColor = 0xFF0000;
            } else {
                this["onboard"+(i+1)].textColor = 0xFFFFFF;
            }
        }
```

Then, we'll show both the score and the number of trash objects still out there for the player to find:

```
        // set number left and score
        numLeft.text = String(trashObjects.length);
        scoreDisplay.text = String(score);
    }
```

Game End

When the game is over, we'll remove the listeners, but not the `gamesprite`. That's because we didn't create the `gamesprite`. It will simply disappear when we use `gotoAndStop` to go to the next frame. Because the `gamesprite` is only on the "play" frame, it will not be shown on the "gameover" frame:

```
    // game over, remove listeners
    public function endGame() {
        blocks = null;
        trashObjects = null;
        trashcans = null;
        this.removeEventListener(Event.ENTER_FRAME,gameLoop);
        stage.removeEventListener(KeyboardEvent.KEY_DOWN,keyDownFunction);
        stage.removeEventListener(KeyboardEvent.KEY_UP,keyUpFunction);
        gotoAndStop("gameover");
    }
```

When the "gameover" frame has been reached, it calls back to `showFinalMessage`. We can't call it earlier because the `finalMessage` text field is only on the "gameover" frame and cannot be accessed until that frame is visible.

We'll place the final time in this text field:

```
    // show time on final screen
    public function showFinalMessage() {
        showTime();
        var finalDisplay:String = "";
        finalDisplay += "Time: "+timeDisplay.text+"\n";
        finalMessage.text = finalDisplay;
    }
```

One last function we need is the `playSound` utility function. It simply serves as a central place for all sound effects to be triggered from:

```
public function playSound(soundObject:Object) {
    var channel:SoundChannel = soundObject.play();
}
```

NOTE
An advantage of having a single function where all sound effects are initiated is that you can quickly and easily build in mute and volume functions. If you placed your sound code all over the game, you'd need to modify each one of those places to add a check for a mute or volume setting.

Modifying the Game

This game can be modified to be almost any free-exploring, item-collecting game. You can change the background elements with no programming at all. And the collision areas, the `Blocks`, can be changed by just moving and adding new `Block` movie clips.

You could even make the game last longer by having more pieces of trash appear as time goes on. You could set a `Timer` so that a new piece of trash is added every five seconds, for instance. The `Timer` could do this for a few minutes before it stops.

You could also add negative items—ones that you want to avoid. These can be things such as oil slicks or land mines. A military version of this game could have a hospital vehicle picking up soldiers on a battlefield, but you need to avoid the land mines.

Building a Flash Racing Game

One thing you might be tempted to do while playing the top-down driving game is to race. For instance, you can see how fast you can make it around the campus.

Although the previous game is a good start, we need to add a few more elements to make a racing game.

SOURCE FILES
http://flashgameu.com
A3GPU12_RacingGame.zip

Racing Game Elements

Even though we are building an "arcade" racing game, and not a real racing simulation, we want to make it realistic enough to make it *feel* like a real car. This means that the car shouldn't lurch into full speed the minute the up arrow is pressed, and it shouldn't stop as soon as the up arrow is released.

We'll be adding acceleration and deceleration to the game. So, the up arrow will add acceleration to the speed of the car. And then, the speed of the car will be used to determine movement for each frame.

NOTE

The distinction between an arcade game and a simulation is bigger here than in any other kind of game we have looked at. A true simulation takes into account physics, such as the mass of the car, the torque of the engine, and the friction between the tires and the road, not to mention skidding.

Not only is this beyond the scope of a simple Flash game, but it is usually simplified or ignored in many high-budget console games. It is important to not let reality get in the way of fun. Or, let it get in the way of finishing a game on time and on budget.

Likewise, if the down arrow is pressed, there will be reverse acceleration. So, from a standstill, the down arrow will produce a negative speed value, and the car will move backward.

Another aspect of a racing game is that the car should follow a specific path. The player shouldn't be able to cut across the track, or reverse over the finish line to cross it again in a few seconds.

To keep track of the player's path, we'll use a simple technique called *waypoints*. Basically, the player needs to get close to and mark off a list of points around the track. Only after the player has hit all of these points will he or she be allowed to cross the finish line.

The best part about waypoints is that players don't even know they are there. We'll hide the waypoints, and quietly mark them off, without bothering the players about this detail. All they know is that they have to race fast and honest.

One last feature of this game that adds a little more of a racing feel is the starting countdown. Instead of the game just starting, there will be three seconds where the player cannot move, and a large 3, 2, and 1 display.

Making the Track

The collision detection in the top-down driving game was based on rectangular blocks. It is fairly easy to detect collisions against straight horizontal or vertical edges.

However, a racetrack includes curves. Detecting collisions against curves, or even short segments of diagonal walls, is much more difficult.

So, we'll avoid it altogether in this game.

The track will consist of three areas: the road, the sides, and everything else. If the car is on the road, it will move unimpeded. If it is on the side of the road, it will still move, but with a constant nagging deceleration that will cause the racer to lose time. If the car is off both the road and the side, the deceleration will be severe, and the car will need to turn and limp back on to the road.

Figure 12.9 shows these three areas. The road is in the middle, and appears gray in Flash. Just outside of it is the side, a brown area, which just appears as a slightly different gray in this black and white figure.

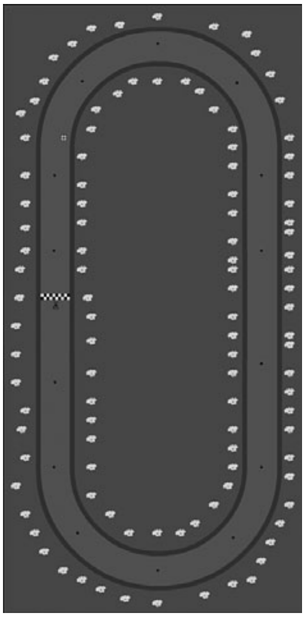


Figure 12.9
The track is surrounded by a thicker "side" element.

The track also includes some nonactive elements, such as the trees you can see scattered around.

NOTE
Although the trees are not referenced in our code, and are not even movie clips, just graphics symbols, they do serve an important role. Without these incidental elements, it is sometimes hard for the player to notice the movement of the car and gauge its speed.

The `car` movie clip is placed on the track in the position where it will start. This happens to be right on the finish line, which is a separate movie clip.

The dots you see around the track are `Waypoints`. You can place only a few around the track, like we have done, or many more if your track includes more twists and turns and you need to prevent the player from cheating and cutting across curves.

All these elements are in the `Track` movie clip, which is the `gamesprite` referred to in our code.

Sound Effects

This game will use quite a few sounds effects. Three different driving sounds will loop as the player moves the car. Here is a list of all the sounds used in the game:

- **`DriveSound`.** A sound loop that plays while the car is accelerating and is on the road. It sounds like a sports car engine.
- **`SideSound`.** A sound loop that plays while the car is accelerating and on the side of the road. It sounds like tires moving through dirt.
- **`OffroadSound`.** A sound loop that plays while the car is accelerating and off both the road and the side of the road. It sounds like a car moving though gravel.
- **`BrakestopSound`.** A screeching-brake sound to be used when the car crosses the finish line.
- **`ReadysetSound`.** A high beep that sounds during the countdown at the start of the game.
- **`GoSound`.** A low beep that sounds when the countdown reaches zero.

The game could easily have more sounds, such as an idle sound for when the car is not accelerating. Also, the `BrakestopSound` could be replaced with a crowd cheering sound as an alternative way to end the race.

Constants and Variables

Some parts of this game's code is the same as the top-down driving game. We focus on the new code here.

The constants now include acceleration and deceleration constants. They are pretty small because they will be multiplied by the milliseconds that pass by between frames:

```
public class Racing extends MovieClip {

    // constants
    static const maxSpeed:Number = .3;
    static const accel:Number = .0002;
    static const decel:Number = .0003;
    static const turnSpeed:Number = .18;
```

The game variables include a `gameMode`, which will indicate whether the race has started. We also have a `waypoints` array, to hold the `Point` locations of the `Waypoint` movie clips. The `speed` variable holds the current rate at which the rate is moving, which will change as the car accelerates and decelerates:

```
    // game variables
    private var arrowLeft, arrowRight, arrowUp, arrowDown:Boolean;
    private var lastTime:int;
    private var gameStartTime:int;
    private var speed:Number;
    private var gameMode:String;
    private var waypoints:Array;
    private var currentSound:Object;
```

Here are the initial definitions for all of the new sounds. Each one is in the library and has been set to export for ActionScript use:

```
    // sounds
    static const theBrakestopSound:BrakestopSound = new BrakestopSound();
    static const theDriveSound:DriveSound = new DriveSound();
    static const theGoSound:GoSound = new GoSound();
    static const theOffroadSound:OffroadSound = new OffroadSound();
    static const theReadysetSound:ReadysetSound = new ReadysetSound();
    static const theSideSound:SideSound = new SideSound();
    private var driveSoundChannel:SoundChannel;
```

Starting the Game

When this game starts, it doesn't need to look for `Blocks`. Instead, it needs to find `Waypoints`. The `findWaypoints` function will do that. We look at it next:

```
public function startRacing() {

    // get list of waypoints
    findWaypoints();
```

The listeners needed are the same as for the top-down driving game, but the variables that need to be set at the start of the game now include `gameMode`, and `speed`. We'll also set the `timeDisplay` text field to empty because it will be blank for the first 3 seconds of the game, until the race starts:

```
    // add listeners
    this.addEventListener(Event.ENTER_FRAME,gameLoop);
    stage.addEventListener(KeyboardEvent.KEY_DOWN,keyDownFunction);
    stage.addEventListener(KeyboardEvent.KEY_UP,keyUpFunction);

    // set up game variables
    speed = 0;
    gameMode = "wait";
    timeDisplay.text = "";
    gameStartTime = getTimer()+3000;
    centerMap();
}
```

Notice that the `gameStartTime` has three seconds added to it. This is because the game will start with a three-second countdown. The car won't be allowed to move until three seconds have passed and the `gameTimer()` catches up with the `gameStartTime`.

The `findWaypoints` function is very similar to the `findBlocks` function in the previous game. However, this time we only need to know the `Point` location of each waypoint. After we record that, the movie clip itself is irrelevant:

```
// look at all gamesprite children and remember waypoints
public function findWaypoints() {
     waypoints = new Array();
     for(var i=0;i<gamesprite.numChildren;i++) {
          var mc = gamesprite.getChildAt(i);
          if (mc is Waypoint) {
               // add to array and make invisible
               waypoints.push(new Point(mc.x, mc.y));
               mc.visible = false;
          }
     }
}
```

The Main Game Loop

We'll skip the keyboard listener functions because they are identical to the top-down driving game.

The `gameLoop`, however, is different. We'll include much more of the game mechanics right inside the function, instead of delegating it to other functions.

After determining the amount of time that has lapsed since the last time `gameLoop` ran, we'll examine the left and right arrows and turn the car:

```
public function gameLoop(event:Event) {

     // calculate time passed
     if (lastTime == 0) lastTime = getTimer();
     var timeDiff:int = getTimer()-lastTime;
     lastTime += timeDiff;

     // only move car if in race mode
     if (gameMode == "race") {
          // rotate left or right
          if (arrowLeft) {
               gamesprite.car.rotation -= (speed+.1)*turnSpeed*timeDiff;
          }
          if (arrowRight) {
               gamesprite.car.rotation += (speed+.1)*turnSpeed*timeDiff;
          }
```

Notice that three factors impact the amount of the turn: the `speed`, the `turnSpeed` constant, and the `timeDiff`. In addition, the `speed` is supplemented by .1. This allows the player to turn the car slightly when at a standstill, and slightly more when moving slowly. Although not accurate to a driving simulation, this does make the game less frustrating to play.

NOTE

By tying the `speed` to the amount the `car` turns, we're allowing the `car` to turn faster when it is moving faster. This makes the steering feel a little more realistic and helps get around the curves.

Also, notice that turning, and the movement to come next, only happens if the `gameMode` is set to "race." This won't occur until the three-second countdown is over.

The `car` movement is dependent on the `speed`. The `speed` is dependent on the acceleration, which occurs when the player uses the up or down arrows. This next bit of code takes care of these changes and makes sure that the `speed` doesn't get too out of control by restricting it to `maxSpeed`:

```
// accelerate car
if (arrowUp) {
    speed += accel*timeDiff;
    if (speed > maxSpeed) speed = maxSpeed;
} else if (arrowDown) {
    speed -= accel*timeDiff;
    if (speed < -maxSpeed) speed = -maxSpeed;
```

However, if neither the up or down arrow is being pressed, the car should slowly come to a halt. We'll use the `decel` constant to reduce the speed of the car:

```
// no arrow pressed, so slow down
} else if (speed > 0) {
    speed -= decel*timeDiff;
    if (speed < 0) speed = 0;
} else if (speed < 0) {
    speed += decel*timeDiff;
    if (speed > 0) speed = 0;
}
```

NOTE

You could also easily add a brake to the car. Just include the spacebar along with the four arrow keys when looking at the keyboard. Then, when the spacebar is pressed, you can have a more severe slowdown than the `decel` constant.

We only need to check the car's movement if there is a `speed` value. If the car is standing perfectly still, we can skip the next part.

However, if the car is moving, we need to reposition it, check whether it is on the road or not, center the map over the car, check to see whether any new `waypoints` have been encountered, and check to see whether the car has crossed the finish line:

```
        // if moving, then move car and check status
        if (speed != 0) {
            moveCar(timeDiff);
            centerMap();
            checkWaypoints();
            checkFinishLine();
        }
    }
```

Whether the car moves or not, the clock still needs to be updated:

```
    // update time and check for end of game
    showTime();
}
```

Car Movement

The car moves depending on the `rotation`, `speed`, and `timeDiff`. The `rotation` is converted to radians, and then fed into `Math.cos` and `Math.sin`. The original position of the car is stored in `carPos` and the change in position in `dx` and `dy`:

```
public function moveCar(timeDiff:Number) {

    // get current position
    var carPos:Point = new Point(gamesprite.car.x, gamesprite.car.y);

    // calculate change
    var carAngle:Number = gamesprite.car.rotation;
    var carAngleRadians:Number = (carAngle/360)*(2.0*Math.PI);
    var carMove:Number = speed*timeDiff;
    var dx:Number = sMath.cos(carAngleRadians)*carMove;
    var dy:Number = Math.sin(carAngleRadians)*carMove;
```

While figuring out where the new location of the car should be, we also need to figure out which sound should be playing. If the car is moving, and it is on the road, `theDriveSound` should be playing. We'll assume that is the case at this point, and adjust the value of `newSound` as we examine more aspects of the game state:

```
    // assume we'll use drive sound
    var newSound:Object = theDriveSound;
```

The first test we perform here is to see whether the car is currently on the road. We'll use `hitTestPoint` to determine this. The third parameter in `hitTestPoint` allows us to test a point against the specific shape of the road. We need to add `gamesprite.x` and gamesprite.y to the position of the car, because hitTestPoint works at the stage level, with stage positions, rather than at the gamesprite level with gamesprite positions:

```
    // see if car is NOT on the road
    if (!gamesprite.road.hitTestPoint(carPos.x+dx+gamesprite.x,
        carPos.y+dy+gamesprite.y, true)) {
```

Note the critically important exclamation point in the previous line of code. The ! means *not* and reverses the Boolean value that follows it. So, instead of looking to see if the car's location is inside the road, we'll be checking to see if it is *not* in the road.

Now that we know the car is not on the road, the next test is to see whether the car is at least on the side of the road:

```
        // see if car is on the side
        if (gamesprite.side.hitTestPoint(carPos.x+dx+gamesprite.x,
                carPos.y+dy+gamesprite.y, true)) {
```

If the car is on the side of the road, we'll use theSideSound rather than theDriveSound. We'll also reduce the speed of the car by a small percentage:

```
            // use special sound, reduce speed
            newSound = theSideSound;
            speed *= 1.0-.001*timeDiff;
```

If the car is neither on the road nor on the side of the road, we'll use theOffroadSound and reduce the speed by a much larger amount:

```
        } else {
            // use special sound, reduce speed
            newSound = theOffroadSound;
            speed *= 1.0-.005*timeDiff;
        }
    }
```

Now we can set the location of the car:

```
    // set new position of car
    gamesprite.car.x = carPos.x+dx;
    gamesprite.car.y = carPos.y+dy;
```

All that is left is figure out which sound to play. We have newSound set to either theDriveSound, theSideSound, or theOffroadSound. If the player is not accelerating at this moment, however, we want to play no sound at all:

```
    // if not moving, forget about drive sound
    if (!arrowUp && !arrowDown) {
        newSound = null;
    }
```

The `newSound` variable holds the proper sound. If that sound is already playing, and looping, however, we don't want to do anything except let that sound continue. We only want to take action if a new sound is needed to replace the current sound.

If that is the case, we issue a `driveSoundChannel.stop()` command to cancel the old sound, and then a new `play` command with a high number of loops, to begin:

```
        // if a new sound, switch sound
        if (newSound != currentSound) {
            if (driveSoundChannel != null) {
                driveSoundChannel.stop();
            }
            currentSound = newSound;
            if (currentSound != null) {
                driveSoundChannel = currentSound.play(0,9999);
            }
        }
    }
```

In addition to the `moveCar` function, we'll also need the `centerMap` function, which is identical to the one in the top-down driving game in the first part of this chapter. This will keep the `car` visually entered on the screen.

Checking Progress

To check the player's progress around the track, we'll look at each of the `Waypoints` and see whether the `car` is close to them. To do this, we use the `Point.distance` function. The `waypoints` array already contains `Point` objects, but we have to construct one on-the-fly with the location of the `car` to compare it to.

I've chosen 150 as the distance needed to hit a waypoint. This is far enough so that the `car` cannot miss a waypoint in the middle of the road, even if it passes the waypoint off to the side. It is critical that you make this distance large enough so that players cannot sneak by a waypoint easily. If they do, they won't be able to finish the race, and they would have no reason why:

```
    // see if close enough to waypoint
    public function checkWaypoints() {
        for(var i:int=waypoints.length-1;i>=0;i--) {
            if (Point.distance(waypoints[i],
                    new Point(gamesprite.car.x, gamesprite.car.y)) < 150) {
                waypoints.splice(i,1);
            }
        }
    }
```

When a Waypoint is encountered, it is removed from the array. When the array is empty, we know that all waypoints have been passed.

This is precisely what checkFinishLine looks for first. If the waypoints array has any items left in it, the player isn't ready to cross the finish line:

```
// see if crossed finish line
public function checkFinishLine() {

    // only if all waypoints have been hit
    if (waypoints.length > 0) return;
```

On the other hand, if the player has hit all the waypoints, then we can assume he is coming up toward the finish line. We'll check the y value of the car to see whether it has crossed the y value of the finish movie clip. If it has, the player has completed the race:

```

    if (gamesprite.car.y < gamesprite.finish.y) {
        endGame();
    }
}
```

NOTE

If you change the map and reposition the finish line, be careful how you test to see whether the car has crossed finish. For instance, if the car approaches the finish from the left, you need to check to see whether the x value of the car is greater than the x value of the finish.

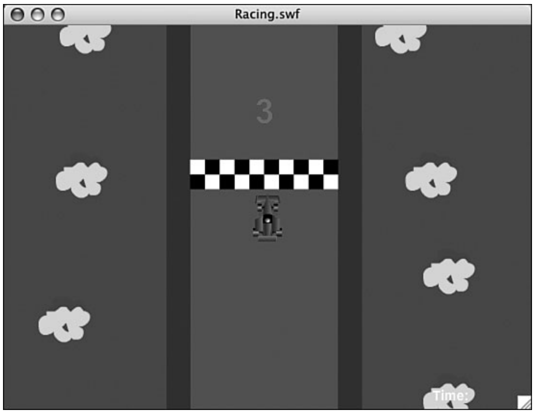
The Countdown and the Clock

Although the clock in this game is very similar to the clock in the top-down driving game, it has a companion clock (in this case, one that counts down the time until the race starts).

If the gameMode is "wait", the race has yet to start. We'll check the gameTime to see whether it is negative. If it is, the gameTimer() has not yet caught up with the three-second delay we created when we set the gameStartTime to getTimer()+3000.

Instead of showing the time in the timeDisplay field, we'll show it in the countdown field. But, we'll only show it as a rounded number of seconds: 3, 2, and then 1. We'll also play theReadysetSound every time this number changes. Figure 12.10 shows this countdown clock at the start of the game.

Figure 12.10
A number in the center of the screen shows the time until the race begins.



```
//update the time shown
public function showTime() {
     var gameTime:int = getTimer()-gameStartTime;

     // if in wait mode, show countdown clock
     if (gameMode == "wait") {
          if (gameTime < 0) {
               // show 3, 2, 1
               var newNum:String = String(Math.abs(Math.floor(gameTime/1000)));
               if (countdown.text != newNum) {
                    countdown.text = newNum;
                    playSound(theReadysetSound);
               }
```

When the `gameTime` reaches 0, we'll change the `gameMove` and remove the number from `countdown`. We'll also play `theGoSound`:

```
          } else {
               // count down over, go to race mode
               gameMode = "race";
               countdown.text = "";
               playSound(theGoSound);
          }
```

For the rest of the race, we'll display the time in the `timeDisplay` field. The `clockTime` function is the exact same one used earlier in this chapter:

```
     // show time
     } else {
          timeDisplay.text = clockTime(gameTime);
     }
}
```

Game Over

When the game ends, we need to do more cleanup than usual. The `driveSoundChannel` needs to stop playing any sound. However, we'll also trigger `theBrakeSound` at this point.

Then, we'll remove all the listeners and go to the "gameover" frame:

```
// game over, remove listeners
public function endGame() {
     driveSoundChannel.stop();
     playSound(theBrakestopSound);
     this.removeEventListener(Event.ENTER_FRAME,gameLoop);
     stage.removeEventListener(KeyboardEvent.KEY_DOWN,keyDownFunction);
     stage.removeEventListener(KeyboardEvent.KEY_UP,keyUpFunction);
     gotoAndStop("gameover");
}
```

After we are at the "gameover" frame, we'll show the final score just like with the top-down driving game. In this case, however, we want to keep the `gamesprite` visible. In the main timeline, it exists across both the "play" and "gameover" frames, so it stays put when we go to the "gameover" frame.

The `showFinalMessage` function is the same as in the previous game, so there is no need to repeat it here. The main timeline also have the same code in the "gameover" frame.

Modifying the Game

The track in this game is pretty simple—just a standard speedway. But, you could make it much more complex, with many twists and turns.

NOTE

The trick to creating first a road and then a roadside movie clip is to just worry about the `road` movie clip first. After you have that perfect, make a copy of it and call that `side`. Then choose the shape inside that movie clip and choose Modify, Shape, Expand Fill. Expand the track about 50 pixels. This will create a copy of the `road` that is thicker and a perfect match the original road.

You could also put hazards on the road. For instance, oil slicks could slow the car down. These could be done the same way as `waypoints`, but the car would have to get very close to them to "hit" them. Then, the `speed` of the car would be affected.

It also common in this type of game to have a dirt patch in the middle of the road. You could do this simply by ripping a hole in the `road` movie clip's shape and letting the `side` movie clip show through.

Another improvement would be to put the `waypoints` in a specific order. Right now, the game ends when the player hits all the `waypoints` and then crosses the finish line. But, the order in which the `waypoints` are hit doesn't matter. So, technically, the player could drive around the track the wrong way, hit all the `waypoints`, and win the minute he hits the last waypoint because he is already above the finish line. But this won't get the player a better time because it takes a while to turn around.

You could order the `waypoints` by naming them something like "waypoint0", "waypoint1", and so on. Then, you would look for each `waypoint` by name rather than by type. Then, only look for the `car` to be near the very next `waypoint`, instead of all of them.

With this racing game, we can see how far we have come. We started with Chapter 3's matching game, a turn-based puzzle game using mouse clicks as input and memory as the skill being tested. We end with a racing game, an action-based animation game using keyboard input and testing a player's dexterity.

This demonstrates the wide variety of games that can be created with Flash. And, if you have been learning from each chapter in this book, it also shows the wide variety of games that you can now build.

The next step is up to you. Modify the games you've created with this book, or start making your own games from your own design. Either way, come participate at http://flashgameu.com if you want to learn more. I hope to see you there.

Index