

Foundation Flash Applications for Mobile Devices

Richard Leggett
Weyert de Boer
Scott Janousek



Foundation Flash Applications for Mobile Devices

Copyright © 2006 by Richard Leggett, Weyert de Boer, Scott Janousek

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

ISBN-13: 978-1-59059-558-9

ISBN-10: 1-59059-558-0

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

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

Java™ and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc., in the United States and other countries.

Apress, Inc., is not affiliated with Sun Microsystems, Inc., and this book was written without endorsement from Sun Microsystems, Inc.

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013.
Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710.

Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit www.apress.com.

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

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

Credits

Lead Editor **Production Editor**
Chris Mills Laura Esterman

Technical Reviewers **Composer**
Marco Casario Diana Van Winkle
Cesar Tardaguila

Editorial Board **Artist**
Steve Anglin, Ewan Buckingham, April Milne

Gary Cornell, Jason Gilmore,
Jonathan Gennick, Jonathan Hassell,
James Huddleston, Chris Mills,
Matthew Moodie, Dominic Shakeshaft,
Jim Sumser, Matt Wade

Proofreaders
Liz Welch
Lori Bring

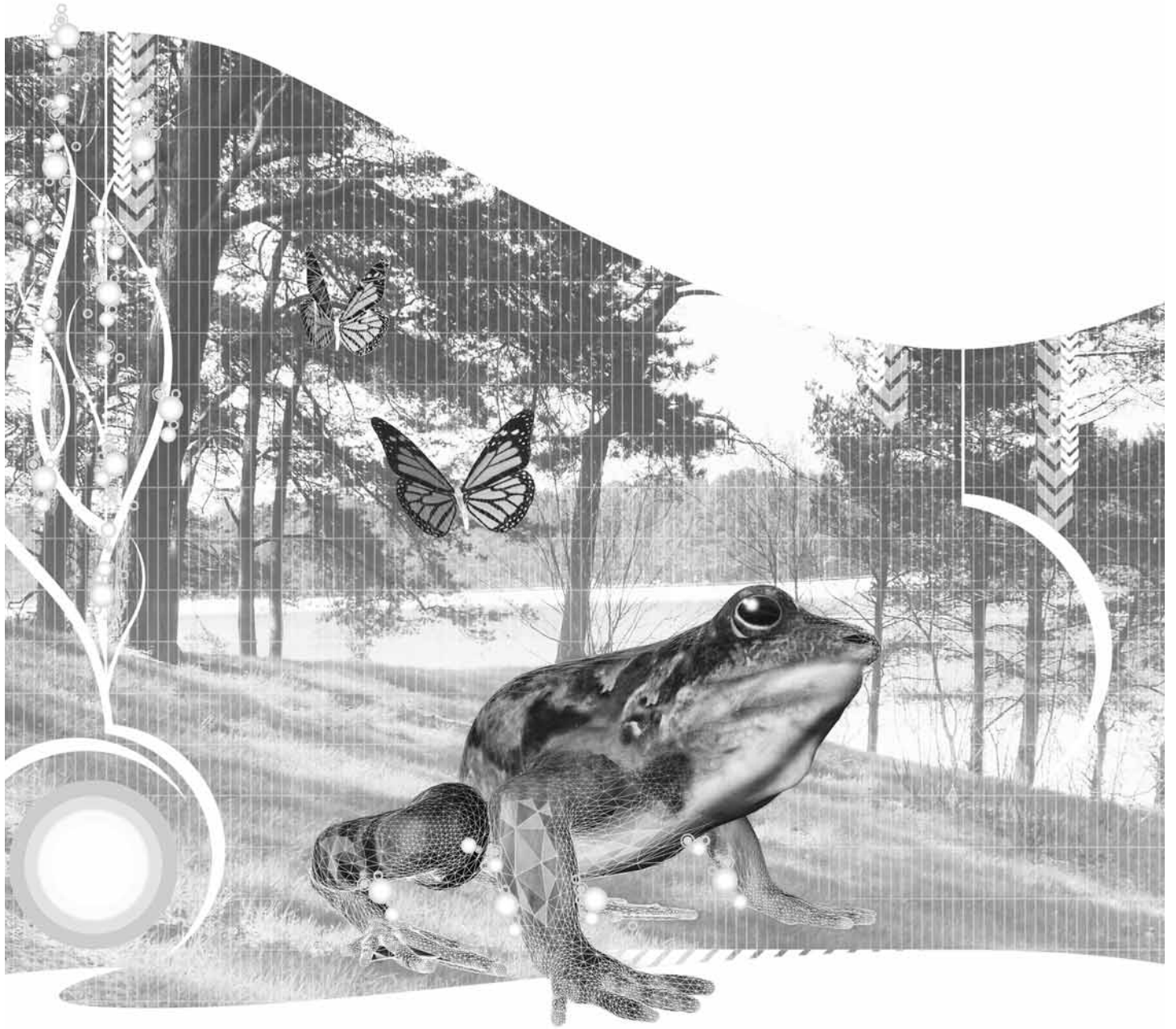
Indexer **Cover Image Designer**
Toma Mulligan Corné van Dooren

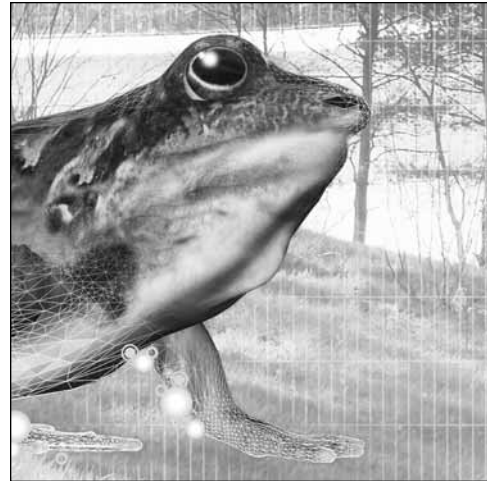
Project Manager | Production Director **Interior and Cover Designer**
Grace Wong Kurt Krames

Copy Edit Manager **Manufacturing Director**
Nicole Flores Tom Debolski

Copy Editors
Heather Lang
Damon Larson

Assistant Production Director
Kari Brooks-Copony





Chapter 6

MOBILE GAMING

Mobile gaming is *big*. Every year, the Game Developers' Conference (GDC) in San Francisco (www.gdconf.com) brings together something in the region of 10,000 developers for a week-long event. In 2005, the GDC opened itself to the mobile gaming arena with GDC Mobile (www.gdconf.com/conference/gdcmobile.htm), an event geared specifically for mobile game developers to teach, to learn, and to discuss the future of the industry. Shortly after that, big players such as Electronic Arts started to look into taking a piece of this pie; EA began developing games for mobile devices and created a mobile division, EA Mobile. By this time, THQ and Midway were already sporting a pair of veteran wings with an acclaimed mobile gaming history, which gives a very good indication that mobile gaming was, and still is, a seriously viable and profitable industry. With all eyes firmly fixed on this sector, the time for early adopters to cash in may have passed, but the demand for games on devices is bigger than ever and continues to rise.

This chapter includes the following topics:

- An introduction to the mobile game scene
- Platforms and game genres
- Moving game elements with code
- The physics of motion
- Player input
- Collision detection and reactions
- Efficient math
- Game assets (including graphics, sound, and video)
- Saving and loading high scores
- Sample games

Introduction to the mobile game scene

Let's begin with no illusions. We are not going to port *Quake III* to Flash for mobile devices any time soon, but we *can* make an endless variety of engaging, educational, exciting, and profitable games that make use of limited resources and can compete well with titles written in any other mobile technology, including Java and even native system code. There are several reasons for this, so I'll expand a little on just how this is possible in the current and future market by first taking a little look back at the evolution of games over the last ten years.

About the summer of 1995, when Sony's PlayStation was released, all of a sudden every game being released seemed to make use of 3-D graphics. Previously, some pseudo-3-D and some basic 3-D were used with isometric games, such as Nintendo's Super FX chip for its Super Nintendo Entertainment System (SNES), and some very clever ray-casting engines, such as the one found in *Doom*. But such raw power had never been used to push hundreds and thousands of dynamically lit, texture-mapped polygons around the screen at over 60 FPS; it was a defining moment for the game industry. With that said, even when it seemed suicidal to release a game that wasn't 3-D in case the public rejected it as old school, 2-D games, such as Team 17's *Worms* series, never stopped being released. Their popularity continued, and some of the best games today are highly original and absolutely two-dimensional. You may be asking yourself, "Where is this going?" Well, when you consider what is possible with Flash and Flash Lite, you need not worry about pure graphical prowess. Your games can be just as popular as those incorporating the latest and greatest graphics; it all depends on the game play and execution.

Tetris is one game that has stood the test of time; it has suffered very little from the countless technological advances since its conception some 18 years ago and is considered by many as the "ultimate" game, ported to every known system, including the mobile phone. When a version of Tetris was released for digital TV services in the United Kingdom, it received one million plays, each costing £0.25 (\$0.44), in the first week alone. A digital set-top box is a limited device, in much the same way a mobile phone is: the processor is slow, the keypad (remote) is awkward, and it is considered a semi-connected device in that it has to dial up via a phone line to make payments. Nevertheless, these 2-D, often very simple games are obviously generating a steady revenue stream for television service operators, and the same mentality can be applied to mobile devices.

Platforms

With our boosted confidence in Flash for mobile gaming, we might want to spend some time scoping out the competition. I'm talking about competition not just in terms of which technology you should use to make your games (I covered that briefly in Chapter 2) but also which devices people will choose to play their games as they move around. The ever-increasing power of phones and PDAs opens up other competition in the mobile gaming space, namely handheld consoles.

There are several types of devices a consumer might choose to play games on. Some options available are PDAs, phones, and handheld consoles such as Nintendo's GameBoy series and Sony's PSP. PDAs offer the option of developing with Java or native operating-system code using C++ (or even C# or VB.NET for the .NET Framework on Pocket PC), as well as with Flash.

PDAs and phones

With PDA and mobile phone platforms, J2ME and native applications generally make up the majority of what is found at present, because until now, these were the only options. Also, J2ME, for one, made it at least somewhat practical to develop for more than one device without a large game studio's budget. Native applications, however, provide the raw power required for these devices to compete with some handheld consoles in terms of graphical complexity. Unfortunately, these devices are still not great for games, as the controls tend to limit the games' ability to make the experience enjoyable for the user. This is improving, however, and one device (indeed, now a series of devices) is aimed specifically at tackling this issue—the N-Gage, later evolving into the N-Series.

N-Gage and N-Series gaming phones

Although some great examples of 3-D games are written in Java, native code is where the speed of newer handsets really shines. With specific math coprocessors boosting performance through the roof, the N-Gage is an example of such a device. Early console games can literally be tweaked to run on a handset with virtually no performance hit. In fact, I've seen Doom 2 run on a mobile device a lot faster than I remember it running on my Pentium PC. But this is where the cookie crumbles, as the technical requirements become higher and higher, fragmentation between device manufacturers increases. With the case of hardware-accelerated gaming, games will experience even more compatibility issues between handsets than you see on the considerably more open architecture of the PC, where you still have constant releases of patches to allow games to work with the wide array of graphics cards now available.

Handheld consoles

The term **handheld console** refers specifically to devices whose primary function is to play games. They may also provide functionality such as movie and MP3 playback and Internet browsing, but fundamentally their form factor is to accommodate game play. At present, the two main leaders in this field are Nintendo's ever-successful GameBoy series and Sony's PSP.

One thing these devices do very well is communicate with their parent systems, such as the PlayStation and Wii; what I am talking about here is *convergence*. Convergence is something that Flash is very good at providing. One thing I like to do with my PSP is control my Xbox Media Center playback utilizing a simple Flash movie that is run in the PSP's web browser. As long as the two devices are on my home wireless network, Flash can be used to good effect in these sorts of situations. The PSP homebrew scene

has taken to Flash on the PSP surprisingly well, porting a lot of good web games to build up a nice catalog of free games. I say “surprisingly well,” because most of the guys producing games had to be hardcore C++ coders to make most of the homebrewed games, so perhaps Flash is helping to let more people get their own stuff out there.

This ease of development is really where Flash shines. We can dramatically increase the return on investment (ROI) because of the significantly reduced development time and human resources involved in creating content. There are absolutely thousands of great Flash games on the Web. If you pick the cream of the crop, port those to PSP (sometimes, porting is not even required), and make these accessible through a web portal page, you’d already have a healthy back catalog for people to enjoy with minimal effort. With WiFi proliferating around cafés, airports, and public transportation, you’re never far from being able to enjoy these.

If you already use Flash as a web designer, perhaps one of the most pleasing aspects of choosing Flash over any other technology for creating mobile applications is that you get instant satisfaction for your effort, because you don’t have to trawl through reams of documentation and SDKs that are entirely platform dependent or learn a great deal of new things—you can apply what you already know.

Game genres

Before we look into which sorts of games work and which simply do not, it might be useful to break down just how people use games on their mobile devices. We can break these people into two reasonably distinct camps: hardcore gamers and casual gamers. With mobile devices, the casual gamers make up the vast majority of the total gamers. This might show a lack of truly great games; more likely, it shows that people turn to their handsets for a quick distraction and little more. On the bus, on the train, or when waiting for somebody, the first thing a person with a little time and no friends to chat to will do quite often is take a phone out and begin tapping away; it’s almost some sort of reflex action. Next time you are on public transportation, look at individuals getting on and sitting down to see just how many reach for their phones to avoid conversation with strangers!

Both hardcore and casual gamers play games that fall into several main genres, though. I’d like to list some typical game genres for your consideration:

- **Action:** These are difficult to pull off because of player speed, but this limitation will improve over time. Example: Space shooters.
- **Role-Playing Games (RPG):** RPG games are great overall; plus, you have the option to make use of MSOs in Flash Lite 2 to save progress. Example: Final Fantasy.
- **Sports:** We aren’t talking 3-D soccer, but strategy and management simulations can do well (especially when used with up-to-date, accurate data feeds). Example: Football Manager.
- **Strategy and Logic:** These games work well; minor variations in strategy games can add value for players without costing too much extra development time. Example: Minesweeper.
- **Skill:** Skill games can work well; we will look at an example of this later in the chapter. Example: Tower of Pisa.

- **Multiplayer:** Multiplayer games offer a potential sweet spot yet to be proven in the mainstream, but phones are, of course, social devices. Example: Mad Bomber.
- **2-D Racers:** These are on par with action games speed-wise. They're technically possible, but limited controls can make these problematic. Example: Micro Machines.
- **Card and Board Games:** These games are great for longevity, and distractions don't affect game play much. They're not so good to just quickly pick up and play. Example: Monopoly.
- **First-Person Shooter (FPS):** Creating these games in Flash Lite is almost impossible, although it has been done with limited success (the results come closer to Wolfenstein in terms of 2-D billboard sprites than true FPS games). Example: Quake.

Regardless of the game type you choose, the key is to find the balance between making the game play as engaging as possible and bludgeoning the CPU to death with a million operations per second. The Flash Player deals with poor CPU treatment by either throwing an error and quitting, or more commonly, running at a fraction of the desired FPS rate and resulting in a less-than-desirable experience.

Code samples

As with previous chapters, this chapter covers both Flash Lite 1.1 and 2.0 ActionScript, on a per-case basis. The aim is to make you comfortable with whichever format you may find in examples, articles, and tutorials online. By now, you should be finding it easier to distinguish between the two, and it is always good practice to be able to, with minimal effort, convert between them where required.

Making things move

“Making things move” is not just the name of an outstanding book on this very subject—of course, I'm referring to *Foundation ActionScript Animation: Making Things Move* by Keith Peters (Berkeley: friends of ED, 2005). I'd like to now briefly look at how you can listen for user input to move a single object or several objects at the same time on screen, and I'd like to look into how this continuous movement might be achieved for the one-button game concept described later in the section entitled “Player input.”

Moving an item directly

Let's try placing a movie clip on stage and moving it every frame; that's 12 times a second at the default 12FPS setting. Fire up Flash, start a new Flash Lite 2 stand-alone mode project, and create yourself a movie clip; it can contain absolutely anything you desire, including our faithful old red ball. Now, place that on stage, and name it `my_mc` in the Properties panel. To get this moving, you can take one of the two approaches that I'll describe in this section. First of all, let's look at assigning a function to the movie clip's `onEnterFrame` loop on frame 1 of the main timeline:

```
my_mc.onEnterFrame = function()
{
    this._x += 1;
}
```

Quite simply, this function moves our movie clip 1 pixel to the right every 1/12 of a second. The preceding code is known as an **anonymous** function; we are assigning something a function that has no name. Alternatively, we could have written

```
my_mc.onEnterFrame = moveMe;

function moveMe()
{
    this._x += 1;
}
```

The important thing to note here is that, just like with the anonymous function, when the function `moveMe()` runs, it runs in the *scope* of `my_mc`; so the keyword `this` refers to `my_mc` itself, and not `_root`, which is where we are actually declaring the function. We could have ten movie clips on stage, all with their `onEnterFrame` assigned to `moveMe()`. They would all move independently, as `this` would refer to each one separately as they each run the `moveMe()` function. The better alternative is to have another object governing the motion of `my_mc` (indeed, all of our ten movie clips), so we can have one loop function that can control the position of lots of clips at once without those clips necessarily all moving at the exact same speed.

Moving several items at once

Let's take our movie clip, and give it a linkage ID in the library (e.g., "ball") so that we can use `attachMovie()` to dynamically add some to the stage:

```
var ball1 = attachMovie( "ball", "ball1", 1 );
var ball2 = attachMovie( "ball", "ball2", 2 );
onEnterFrame = function()
{
    ball1._x += 1;
    ball2._y += 1;
}
```

Here we have our two ball instances moving independently, controlled by just a single `onEnterFrame`. In this case, it belongs to the main timeline, `_root`, which is useful for several reasons. Not only could we compare positions of the ball instances to each other in this one loop (for example, in collision detection between them), but if we had 100 balls, bullets, or sprites of any kind, giving each one its own `onEnterFrame` would seriously bog down the CPU. Having one `onEnterFrame` to govern them all is a more efficient way of tackling animation and can make for easier-to-follow code.

Moving items around a point

Now you can animate objects moving in two directions, you can expand your capabilities a little. Let's think about some situations we might need to model in a game. Take orbiting, for example, where one or more objects orbit another. An example of this might be a shield made up of several sprites that rotate around the player's ship. If you'd like to follow along with this example in Flash, open `RotateAroundAPoint.fla`. You should find the following code on frame 1:

```

var orbitPos = 0;

onEnterFrame = animate;
function animate()
{
    var xPos = Math.sin( orbitPos * Math.PI/180 ) * 50;
    var yPos = Math.cos( orbitPos * Math.PI/180 ) * 50;

    planet_mc._x = xPos + star_mc._x;
    planet_mc._y = yPos + star_mc._y;

    orbitPos += 2;
    if( orbitPos > 359 ) orbitPos = 0;
}

```

We have another familiar `onEnterFrame` loop here. This could also be written `this.onEnterFrame = animate;`, but I chose to leave out the explicit scoping of the variables and function in question to give you a feeling for how Flash figures it out by following some standard rules. This time, the aim is to orbit `planet_mc` around `star_mc`. The first thing to do is define a variable to store the current orbit position, `orbitPos`; this is how far around the star the planet has orbited, ranging from 0 to 359 degrees. At the end of the loop, you will notice that we are incrementing this value by 2 each frame, until it reaches anything over 359 degrees, at which point, the orbit resets, as the planet must have come full circle!

Our next step in the loop is to work out the next x position and y position of our planet. We do this by multiplying the distance away from the planet, a whole 50 pixels(!) by the sine or cosine of the orbit angle in radians for x and y, respectively. The multiplication by `Math.PI/180` just converts degrees to radians, which the sine and cosine functions expect.

The second block of code in the `onEnterFrame` function applies these values to the `_x` and `_y` properties of the planet, also adding the `_x` and `_y` of the star, so that we are, indeed, rotating around that and not just the point (0, 0) at the top-left corner of the screen.

The physics of motion

Now that you can move things around at a constant speed, you might want to branch out a bit and look at how things move in the real world. In reality, it's almost impossible for something to remain still (on microscopic and quantum levels, nothing is ever *completely* still). But we can assume that, for our purpose, every object has a certain velocity ranging between negative infinity and positive infinity, including zero. Notice that I use the term velocity instead of speed, because a *velocity* can be negative as well as positive, for example, when reversing a car instead of driving forward.

Velocity, direction, and momentum

With velocity, we can also take into account in which direction an object is traveling. For example, I might have a pedometer that tells me I'm running at 12mph down a windy road. At the same time, it might be said that, because I'm running toward the northeast, overall I'm actually traveling at, say, 7mph in a northerly direction and 10mph in an easterly direction. I'm still going 12mph toward the northeast, but

we can break it down into these individual directions also. In a 2-D game, we might represent this breakdown with an object's x velocity (from left to right) and y velocity (from top to bottom). Breaking down what we know as a single value for velocity into individual velocities in certain directions is also known as breaking it down into its **components**.

*In math, when we take a velocity and a direction, what we get is usually called a **vector**. A vector can be thought of as a line that has both magnitude (e.g., length or speed) and direction (e.g., an angle). We can use some simple math to do things like add up 100 vectors that represent the speed and direction of 100 particles within a sphere to get the overall speed and direction for that sphere in its entirety.*

Let's take a look at how we might apply velocity to a movie clip in Flash Lite 2. You can follow along with this one using `Velocity.fla`. First of all, create a movie clip on stage, and give it the instance name `my_mc`. Next comes the code for frame 1 of the root timeline:

```
my_mc.xVel = 3;
my_mc.yVel = 0;
my_mc.onEnterFrame = animate;
function animate()
{
    this.xVel -= 0.11;
    this.yVel += 0.1;

    this._x += this.xVel;
    this._y += this.yVel;
}
```

Flash Lite 1.1 developers take note! This Flash Lite 2 code requires minimal modification to work in Flash Lite 1.1. Instead of writing the `onEnterFrame` function as it appears in the preceding code snippet, simply create a two-frame loop inside the `my_mc` symbol and place the code within one of those frames instead of in the previously-defined function. This is covered in Chapter 3, so feel free to jump back if you need a refresher on creating code loops in Flash Lite 1.1.

OK, so that example is fairly boring. We can see the ball curving back on itself as we alter its velocity in the x and y directions and apply those values to each frame. You can think of `-= 0.11` and `+= 0.1` as applying acceleration (or deceleration) to the object's velocity. Now, you can start to think about how you can control not only the direction something is traveling in but also how fast it is accelerating in that direction. We'll look at a real-world use for this with the jumping sheep example a little later on. For now, I'd like to look at one more useful way of animating objects with code.

Creating an ease out tween

One of the most common ways of animating a movie clip from one position to the other is with the ease out tween. This is available from the motion tween options in the `Properties` panel when animating on the time line, but here, we are going to see how it is done with code. You might recognize an ease out without knowing its name; it's characterised by a rapid initial speed that slows to a full stop upon reaching the target. Modifying the previous code, we get the following ease out code:

```
endX = 100;
endY = 100;
my_mc.onEnterFrame = function()
{
    this._x = this._x + (endX-this._x)/10;
    this._y = this._y + (endY-this._y)/10;
}
```

You may notice I haven't used the `var` keyword to declare my variables. You can and should use this for Flash Lite 2, as it clearly defines in which scope a variable is being created. For the purpose of this example, I'm leaving it off just to show you that you can still write code in a Flash Lite 1.1 style with only minor differences. Going through the code, you can see we have set up a couple of variables to determine the target x and y positions for our movie clip to reach. Inside the `onEnterFrame` loop, we simply move toward that target by following a simple equation:

$$\text{New X Position} = \text{Current X Position} + \frac{((\text{Target X Position} - \text{Current X Position}) / \text{Divisor})}{10}$$

If you break down the equation, you can see that our new x position is the sum of the current x position plus the difference between the target x and the current x position, divided by a value I've called `Divisor`. By taking the target x position and subtracting the current x position, you get the distance you need to move the object. That distance starts off great, and as the object gets closer, diminishes in size. `Divisor` is there to stop you from reaching the target position all at once. The larger the value, the slower the target position is reached; a lower value, thus, increases the speed. The magic happens as the distance to the target reduces. We are still using the same divisor, but it is operating on increasingly smaller values, leaving the object less distance to travel each time, until the value eventually gets near enough to 0. This slows the object as it reaches the target. We can apply exactly the same equation to the y position and, thus, have the ability to ease out to any location we like.

Player input

A lot of the stuff covered in this chapter is purely code based, things that you can just leave running to show various game-related concepts. But it's good to remember that in games all of these things are directly related to some sort of user input. This section will look at taking that user input and doing something with it. I've mentioned before that the keys on most mobile devices aren't the best for gaming; they're usually small and fiddly. Let's now talk about some ways to make sure this isn't too much of an issue.

One-button games

That may sound like a crazy claim to make, but it is possible to make games that have just a single button for all user input. In fact, you can find a whole lot of these designed by the game gurus over at globz.com. With only one button for input, you have to rely on one thing to make the game interesting—*time*. Effectively, this sort of game relies on something automatic occurring as time passes, and the user presses a button to interact at given points. Take, for example, a base jumper jumping out of a building. As she falls, the user sees windows rushing past and has to choose when to open the parachute; the further the jumper drops, the more points the user gets, but if she drops too far, she'll crash into the ground and have a really bad day. It's true the user might also need another button to access the menu or quit the game, but what we have here is just one example of how you can use time to make up for the poor buttons available on phones, which make it difficult to perform rapid button presses in games.

This same concept is good for another Flash Lite limitation—the inability to detect a keyUp event. Normally, we'd listen for the keyDown, start moving a character, for example, and stop moving her on the keyUp (alternatively, we could use `Key.isDown()` to check whether a key is still being depressed). Without being able to listen for a keyUp or check `Key.isDown()`, we have no way of knowing when to stop the character, so any movement must, instead, be done by individual increments every time you press the key. This makes for jerky and delayed movement. If the character can be allowed to constantly move (perhaps using an `onEnterFrame` loop), we can listen for keyDown events to change the direction of that motion. This constant movement brings the feeling of accurate control over an on-screen element.

Using input with motion in a jump example

It's time to apply some of what you've learned. In this example, we are going to take a simple side-on hero character, something like Mario from the Nintendo game series, but in this case, we will use a sheep to avoid any nasty copyright issues. We are going to get him running and jumping with key presses, and we are going to take the constant movement approach, so when you release the key he will keep running unless you change direction, jump, or hit a wall.

To follow along please open `jump.fla` from the Chapter 6 examples. The screen for `jump.fla` is shown in Figure 6-1.

There's not a lot going on here. We have some dummy background graphics, and our hero movie clip containing a sheep, named `hero_mc`. All of our code is on one frame as usual, to make it easier to explain. With practice, you will find that you prefer to separate code into separate files where possible, using some of the techniques described in Chapter 4. For now, let's concentrate on the raw code. There's quite a bit this time, so I'm going to go through it in segments, from top to bottom. The first segment follows:

```
var direction = "";
var gravity = 0.8;
var yspeed = 0;
var floorHeight = hero_mc._y;
hero_mc.gotoAndStop(1);
```

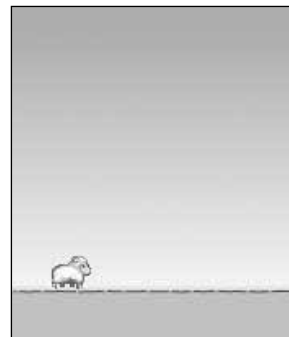


Figure 6-1. The jumping sheep example

You should be fairly familiar with the process of setting up the scene at the start. We create a couple of variables, the first being the initial direction of the hero (which we initialize as a blank string, as the user has yet to press a directional key). The next two variables are part of our jump code, and we will look at those later. Next, we measure the vertical position of the hero character, so we know where the floor is (assuming the hero is on the floor when the game starts, based on where we position him on stage). Finally, we tell `hero_mc` to go to frame 1 and stop using `gotoAndStop()`. This prevents the walking animation from occurring while the character is not moving at the start. If you drill down into the `hero_mc` movie clip, you can see a simple time line with just enough frames to give the impression of walking animation when played back, so this `gotoAndStop()` action prevents this loop from playing through at the start. Next, we have the key listener code on frame 1:

```
// Create listener object
var keyListener = {};

// Add listener object to the Key object's listeners array
Key.addListener( keyListener );

// Define the onKeyDown event called when a user presses a key
keyListener.onKeyDown = function()
{
    // Check which key they pressed
    if( Key.getCode() == Key.RIGHT )
    {
        direction = "right";
        hero_mc.gotoAndPlay("walk");
    }
    else if( Key.getCode() == Key.LEFT )
    {
        direction = "left";
        hero_mc.gotoAndPlay("walk");
    }
    else if( Key.getCode() == Key.UP )
    {
        if( hero_mc._y >= floorHeight ) jumpPressed = true;
    }
}
```

The usual practice is to create a simple object to listen for any keypresses (in this case, called `keyListener`). Then, add it as a listener to the `Key` object itself. This means that whenever the user presses a key, a function named `onKeyDown` will be executed on any listening objects, so we must define `onKeyDown` in order to intercept these keypresses. Within this function, we make use of the `if` statement to check the key code for whichever key was pressed, for example, `Key.RIGHT` or `Key.LEFT`.

Let's take each block of this triple `if` statement in turn. The first checks to see if the user pressed the right button. If so, we simply set the direction to "right", and start the walk animation contained within `hero_mc`. You may wonder what could possibly change just by setting this direction variable to a string—all will be revealed in the game loop, which we will come to shortly. If the user did not press right, we check the same for `Key.LEFT` and, accordingly, set the direction to "left". If neither of these keys were pressed, we check whether the up key, `Key.UP`, was pressed, and quite simply, we set

another variable `jumpPressed` to `true`, but only if `hero_mc` is not above the value stored for the ground level, `floorHeight`. Now, let's move on to the penultimate piece of the puzzle, the game loop:

```
onEnterFrame = function()
{
    if( direction == "right" )
    {
        if( hero_mc._x+hero_mc._width/2 < Stage.width )
            hero_mc._x += 2;
        else
            hero_mc.gotoAndStop(1);

        hero_mc._xscale = 100;
    }
    else if( direction == "left" )
    {
        if( hero_mc._x-hero_mc._width/2 > 0 )
            hero_mc._x -= 2;
        else
            hero_mc.gotoAndStop(1);

        hero_mc._xscale = -100;
    }

    if( jumpPressed )
    {
        yspeed = -8;
        hero_mc.onEnterFrame = jump;
        jumpPressed = false;
    }
}
```

The first part of this `onEnterFrame` (game) loop takes into account that variable we set when the user presses left or right, `direction`. If the direction is "right", the game loop does two things. The first is to check whether our hero is hitting the right-hand side of the screen with `if(hero_mc._x+hero_mc._width/2 < Stage.width)`. This evaluates to `true` if the center point of our hero graphic is at or past the right-hand edge of the screen. If not then, the `_x` position is incremented by 2 pixels. Also, the `_xscale` is set to positive 100; this may already be the case, but essentially, this just makes our hero face to the right. If we find that the hero is past the edge of the right of the screen (even by just half a pixel), the running animation stops and his `_x` position is not incremented. The next part of the code just repeats this for the "left" direction, but instead decrements `_x` and sets `_xscale` to -100, flipping the character.

Now, let's look at the jump code. If we find that the `jumpPressed` variable is set to `true` (meaning the user has pressed `Key.UP`), we set our `yspeed` variable to -8; initially, this was 0. We will use `yspeed` to actually move our hero along his path up and back down to earth just by changing this value. Next, we are setting `hero_mc`'s `onEnterFrame` function to point to another function, named `jump`. Finally, we revert `jumpPressed` to `false`, so that this code doesn't execute again until the user presses the jump key another time.

OK, we are almost there, but we have one last bit to examine. In the game loop, you will remember we set our `hero_mc`'s `onEnterFrame` to a function called `jump`. This is the last piece of code we need to get the jumping in there:

```
function jump()
{
    yspeed += gravity;
    this._y += yspeed;

    if( this._y >= floorHeight )
    {
        yspeed = 0;
        this._y = floorHeight;
        delete this.onEnterFrame;
    }
}
```

If you look back, you can see that we set `yspeed` to `-8` when the user pressed the up key. Now, we are going to make use of that variable, adding to it our arbitrary value for the effect of gravity, which we set to be `0.8` at the start. This will bring `yspeed` up from its initial `-8`, closer to `0` and beyond, each time this loop runs through. Now you will see why we did this. In the next line, we increment `this._y` by `yspeed`. It's important to note that, because we set the `jump` function to the `onEnterFrame` of `hero_mc`, `this` will refer to `hero_mc` itself. So the first time this is code run, `hero_mc._y` will have added to it `-8` plus `0.8`; in other words, we're raising the character vertically by `7.2` pixels. That will make our character jump, and when `yspeed` eventually becomes positive, fall back down to earth. But there's nothing in there yet to stop him from passing right through the floor.

The final bit of code here does just that. We check to see if `this._y` is greater than or equal to `floorHeight`. Remember, in Flash, `_y` increases toward the bottom of the screen. If `this._y` is greater, we know we overshot the floor or at least met it, so we can stop the movement by resetting `yspeed`, positioning `hero_mc` exactly at floor level and deleting the `onEnterFrame` to prevent the jump loop from continuing to execute until next time. Phew, such a simple example takes quite some explanation, but hopefully running through it a few times yourself should help it sink in nicely.

We also touched on our next topic briefly here, when we prevented our sheepy hero from passing the walls of the screen; the next topic is collision detection.

Collision detection

Collision detection plays a vital role in many types of games—not just action games, in which we want to see if maybe a bullet has hit a target, but also in platform games, in which we want to see if our character is standing on a surface or jumping in the air, and in many more situations where we have multiple objects on screen at any time. The subject of collision detection has been covered in many books, but with the added limitations of Flash 4 ActionScript with Flash Lite 1.1, we need to take another look at just how this can be achieved. There are many types of collision detection to fit the various situations you might encounter when making a game. In this section, I try to cover some of the main methods that best suit mobile gaming where processing speed is limited, this includes several options for detecting collisions between objects using both Flash's built-in mechanism and math alone.

MovieClip.hitTest() (Flash Lite 2.0+)

If you are looking to develop for Flash Lite 2, you can take advantage of the very useful `MovieClip.hitTest()` function introduced in Flash 5. The following code shows this function being used to detect whether two movie clips are overlapping, or as we refer to it in games, colliding. Should `mc2` overlap `mc1`, `bang!` will be displayed in the output window:

```
if (mc1.hitTest(mc2)) trace ("bang!");
```

For a moment imagine that `mc1` was a player, and `mc2` was, maybe, some hot coals. We might have this code running every other frame in a frame loop, and if the condition evaluates as true, we could deduct some health from a player's health gauge, for example. `hitTest()` will, by default, check if the objects' bounding boxes are overlapping. The bounding box is an imaginary rectangle that is just large enough to encompass the entirety of the graphics contained in a movie clip. The problem with this is that a circular or irregularly shaped movie clip might register a collision with another even if the visible shape itself isn't touching, because their rectangular bounding boxes *are* still overlapping, as shown in Figure 6-2.

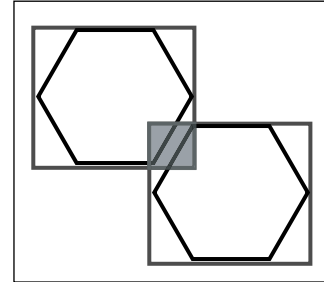


Figure 6-2. Overlapping bounding boxes around hexagons

The solution is to use the second form of the `hitTest()` function, which allows us to set a shape flag, telling Flash to examine the actual outline shape of the movie clip when testing for a collision. The only drawback here is that with our new accuracy, we must test the shape collision against a single point as opposed to two separate objects, or the Flash Player would probably crash because of all of the necessary calculations involved in detecting every possible point of contact along the two outlines. This form of `hitTest()` is illustrated in Figure 6-3.

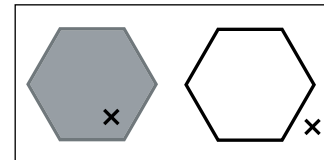


Figure 6-3. Two hexagons showing point-based hit tests

The hexagon on the left in Figure 6-3 shows that the hit test is true, because the point is inside its shape boundary, unlike the one on the right. The code to detect whether our point is anywhere inside the shape follows:

```
if( shape_mc.hitTest( point.x, point.y, true ) )
    trace("point x,y is within shape");
```

This is great for detecting whether a crosshair is over an irregularly shaped target, for example. Check out the `Shape Burst` sample from the Chapter 5 downloads to see this in action (`ShapeBurst.fla`). The game (shown in Figure 6-4) isn't much fun, as there is no way to win, but it *does* illustrate how to perform `hitTest()` checks on a regular basis within a game.

With Flash Lite 1.1, we do not have the luxury of the `hitTest()` function. This is a drawback, but not a fatal one, when it comes to collision detection in games, as we are about to find out.

The registration point

Flash uses something called a **registration point**, an invisible point indicated by the crosshair you see on stage when editing a movie clip or button symbol, for its `_x` and `_y` properties at runtime. It's important to always check the position of the registration point, because it may throw some of the calculations off in the remaining parts of this section. Unless otherwise stated, I will be positioning the registration point of a movie clip directly at its center, or as close as possible for irregularly shaped objects. Once graphics have been converted to symbols, it isn't possible to move a registration point; instead, you move the content around the point until it sits in the right place. Say, for example, you needed to move a registration point to the tip of a missile graphic, so that when performing a `hitTest()` check on a player against a point, that point is at the very front of the missile and will be the first thing to hit the player. To do this, you'd have to edit the missile clip by double-clicking it in the library and moving the shapes or bitmaps that make up the missile to make sure the tip resides precisely on the registration point, as shown in Figure 6-5.

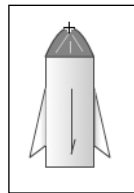


Figure 6-5. Missile showing the position of the registration point (at the top, in the center)

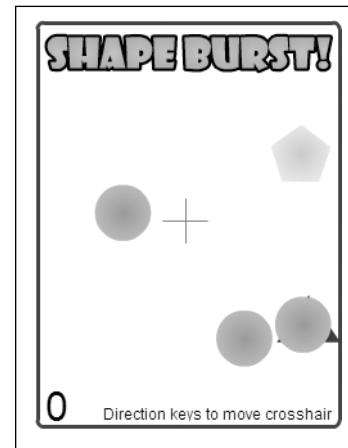


Figure 6-4. The Shape Burst example

Bounding box collisions (Flash Lite 1.1)

Without `hitTest()` to rely on in Flash Lite 1.1, we have to manually check the boundaries of a movie clip to perform collision detection. The properties we need are `_x`, `_y`, `_width`, and `_height`. With these four properties, we can calculate a bounding rectangle like so:

```
mcTop = mc._y - mc._height/2;
mcBottom = mc._y + mc._height/2;
mcLeft = mc._x - mc._width/2;
mcRight = mc._x + mc._width/2;
```

Those four variables mark the boundaries that can then be compared with another object's to see if there is any overlap. Let's rename our movie clip from `mc` to `mc1`, and imagine we have another movie clip named `mc2` on screen that can be moved around with the arrow keys. Our movie might look like the one in Figure 6-6.

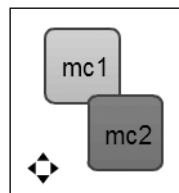


Figure 6-6. Bounding box example

Flash's coordinate system starts from the top-left corner of the screen and the y axis increments as you move down the screen. Therefore, a movie clip with _y property of 50 will appear vertically below a movie clip with _y equal to 10. The x axis, on the other hand, starts at the left and increments to the right as with many other coordinate systems.

We can go about testing for an overlap by running code that examines these two movie clips' boundaries every frame, with the simple two-frame loop mechanism that we used previously. Here is how it is done in Flash Lite 1.1:

```
isColliding = false;

mc1Top = mc1._y - mc1._height/2;
mc1Bottom = mc1._y + mc1._height/2;
mc1Left = mc1._x - mc1._width/2;
mc1Right = mc1._x + mc1._width/2;

mc2Top = mc2._y - mc2._height/2;
mc2Bottom = mc2._y + mc2._height/2;
mc2Left = mc2._x - mc2._width/2;
mc2Right = mc2._x + mc2._width/2;

topOverlap = (mc2Bottom >= mc1Top && mc1Bottom >= mc2Bottom);
bottomOverlap = (mc2Top <= mc1Bottom && mc1Top <= mc2Top);
leftOverlap = (mc2Left <= mc1Left && mc2Right >= mc1Left);
rightOverlap = (mc2Left <= mc1Right && mc1Left <= mc2Left);

if (rightOverlap && bottomOverlap || rightOverlap && topOverlap ||
    leftOverlap && bottomOverlap || leftOverlap && topOverlap)
    isColliding = true;

if (isColliding) tellTarget("mc1") gotoAndStop(2);
else tellTarget("mc1") gotoAndStop(1);
```

That's quite a bit of code just to get a simple bounding-box collision test going, but the odds are you will only be checking one particular object against a few others, in which case you can put this code in a simple for loop and make use of the eval() command (or use the array access operators in Flash Lite 2) to check the movie clip's boundaries against the other movie clips. To view this code in action, open BoundingBox.fla from the Chapter 6 downloads. If you are using ActionScript 1 or 2 in Flash Lite 2, you can simplify things a lot with the hitTest() command mentioned previously. Still, there is an easier way to get basic collision detection into your games, and the next example we are going to look at is perfect for just that.

Circle-to-circle(s) collisions (Flash Lite 1.1+)

One of the quickest and cleanest methods for collision detection in games is to use bounding circles to test for an overlap. A circle is often a good shape to fit the outline of a variety of objects, from balls, to helicopters, and the math involved is actually a lot simpler than when dealing with rectangles!

In Figure 6-7, you can see a pool table with a few balls left to sink.



Figure 6-7. A pool table for the circle-to-circle collision example

If we imagine that the ball on the far right (let's call it ball1) is traveling towards the ball directly in the center (say, ball3), we can work out the distance between them with the following equation:

$$c = \sqrt{a^2 + b^2}$$

Many of you will instantly recognize this as the Pythagorean theorem, which is used to work out the length of the hypotenuse, c , of a right-angled triangle with side lengths a and b . While our triangle isn't visible in the game, it is still there in the math. Side lengths a and b are nothing more than the vertical and horizontal distances between our two objects, and c is the distance between them. We can apply Pythagoras's theorem in ActionScript by working out the vertical and horizontal, and slotting those figures into the previous formula as follows:

```
vertDist = Math.abs(ball1._x - ball2._x);
horizDist = Math.abs(ball1._y - ball2._y);
totalDist = Math.sqrt(horizDist*horizDist + vertDist*vertDist);
```

There you have it—the distance between the centers of ball1 and ball3 are slotted into the equation in the third line, giving us the total distance between the balls. Now, to work out if these balls are colliding, we just need to see if they are closer than the sum of their radii (the radius stretches from the center of the ball, to any point along the bounding circle):

```
radius = ball1._width/2; // assume the same for ball2
if (totalDist <= radius*2) trace("balls are colliding");
```

The balls in the previous example fit with the idea of having a bounding circle that matches exactly the outline of the shape that we are performing the collision test on. But we can also use circles to detect collisions between objects with more-complex shapes, such as vehicles. An example can be found in `JetFighter.fla` (see Figure 6-8).

Figure 6-8 shows the unlikely situation of a jet fighter and a helicopter that are about to collide head on. You might notice that the jet fighter has not one but two circles over it. By using several bounding circles, we can perform more-accurate collision detection with minimal cost to the execution speed of our game, when compared with more-detailed techniques, such as detecting collisions on a per-pixel basis. Using multiple circles allows for more-realistic collisions as the player weaves in and out of enemy fire and ships, minimizing the occurrence of some very frustrating collisions that tend to happen in the unused white space around graphics that also lie within the circular or rectangular boundaries.

The dotted circles in the illustration are actually child movie clips of the vehicle clips, so everything stays together when moving the vehicles around the screen. The helicopter movie clip is named `heli`, and its bounding circle is named `circ`, whereas the jet fighter is named, unsurprisingly, `jet`, with bounding circles `circ1` and `circ2`. What we need to do is check the bounding circle of the helicopter against each of the circles that surround the jet to see if we have any overlaps. We can do this with a simple for loop:

```
isColliding = false;

// loop through 2 circles
for (i=1; i<=2; i++)
{
    heliRad = heli.circ._width/2;
    jetRad = eval("jet/circ" add i)._width/2;

    heliX = heli.circ._x + heli._x;
    heliY = heli.circ._y + heli._y;

    jetX = eval("jet/circ" add i)._x + jet._x;
    jetY = eval("jet/circ" add i)._y + jet._y;

    horizDist = Math.abs(heliX - jetX);
    vertDist = Math.abs(heliY - jetY);
    totalDist = Math.sqrt(horizDist*horizDist + vertDist*vertDist);

    if (totalDist <= heliRad+jetRad) isColliding = true;
}
```

There are more lines of code there than are strictly necessary, because I've expanded certain things like the calculation of `heliRad`, which I could have used without creating an extra variable by placing the code for the helicopter radius directly in the final `if` statement. But I wanted to lay it out like this to make sure it was as easy to follow as possible. The other thing to note is that this isn't the most efficient

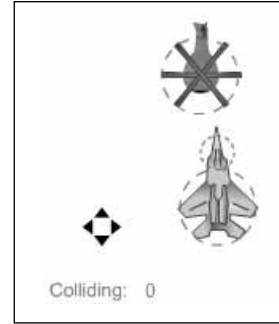


Figure 6-8. A jet plane and a helicopter about to collide

code, as everything is evaluated several times inside the for loop. For example, I could have assigned a value to `heliRad` outside of the for loop, as it will never change, whereas `jetRad` will change, depending on which circle, `circ1` or `circ2`, you are looking at. I've put together a little sample that you can run to see just how this works in `JetFighter.fla` in the Chapter 6 downloads.

As a side note, you can also use circle-to-point collisions. You can easily substitute one of the circles for a point, if you want to see whether any object that's actual size is unimportant—something like a projectile, crosshair, or particle—is inside a circle. To do this, simply check whether the total distance is less than or equal to the radius of the one circle that concerns us, the target, for example, `if (totalDist <= jetRad) trace("Hit by bullet")`.

If you are using Flash Lite 2, you might be more familiar with the use of the array access operators in place of the `eval()` function. You might also be aware that we don't need the physical two-frame loop; instead we can use a simple `onEnterFrame` function loop:

```

this.onEnterFrame = function()
{
    var isColliding = false;

    // loop through 2 circles
    for ( var i=1; i<=2; i++)
    {
        var heliRad = heli.circ._width/2;
        var jetRad = jet["circ"+i]._width/2;

        var heliX = heli.circ._x + heli._x;
        var heliY = heli.circ._y + heli._y;

        var jetX = jet["circ"+i]._x + jet._x;
        var jetY = jet["circ"+i]._y + jet._y;

        var horizDist = Math.abs(heliX - jetX);
        var vertDist = Math.abs(heliY - jetY);
        var totalDist = Math.sqrt(horizDist*horizDist +
                                   vertDist*vertDist);

        if (totalDist <= heliRad+jetRad) isColliding = true;
    }
}

```

As you can see, there's not much of a difference between the code here, and the code listed previously, but hopefully, this side-by-side comparison shows that using dot syntax in Flash Lite 1.1 can produce quite similar code to the more-modern ActionScript 1 and 2. This is particularly true when the code is written in the IDE where you can get away with not specifying `var` to declare variables (although it is recommended to avoid confusion about where the variable is declared!). This would not have been possible back when Flash 4 was out; the IDE improvements since then make it feasible.

Line-to-circle collisions (Flash Lite 1.1+)

Think back to our pool table example; we don't just have collisions occurring between the balls (circle-to-circle collisions). We also have collisions between the balls and the cushions (line-to-circle collisions) that we need to perform some collision checking on, so that the balls don't simply float off the edge of the table. If you have some game programming experience already, you will know that this isn't so tricky; we just see if a ball reaches a certain `_x` or `_y` position and bounce it back in the opposite direction by switching its vertical and/or horizontal velocity. So to make it a little more difficult for ourselves, let us imagine a situation where the lines *aren't* perpendicular or creating a perfect box. Instead, let's have our lines at various angles, such as those found on a miniature golf course (see Figure 6-9).

We can't just check whether the `_x` or `_y` position has reached a certain value now. We need to come up with another way to see if the ball is touching (or passing through) a point on the line—we need to employ a little trigonometry to solve this one.

Get your graph paper ready

Take a look at Figure 6-10, which shows a simple line drawn on a graph.

In addition to drawing the line alongside a pair of axes, we can define it with the following equation:

$$y = mx + b$$

Now, you may remember plotting lines on graph paper at school, or for those lucky enough, entering an equation into your calculator to have it magically drawn for you. Either way, recall that this general equation allows us to describe any straight line that we might want to represent. The `x` and `y` variables are fairly self-explanatory: `y` represents the given vertical position of a point along the line, depending on the values to the right of the equals sign, and `x` relates to the horizontal position depending on `y`. The `m` represents the **gradient** of the line; the gradient is simply the slope of the line. You can get a value for that by first imagining that along our line we have a right-angled triangle, as shown in the triangles in Figure 6-11.

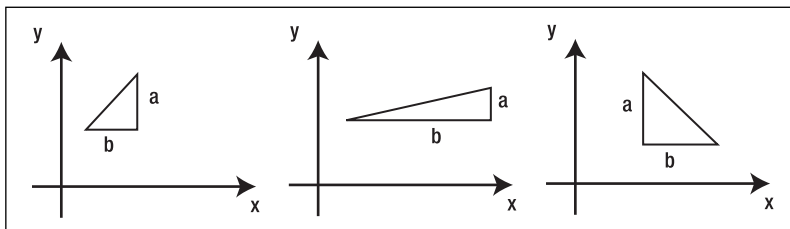


Figure 6-11. Three triangles plotted on graphs

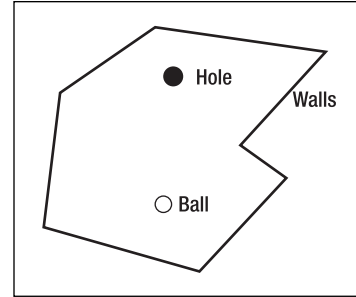


Figure 6-9. A hole for an imaginary miniature golf course

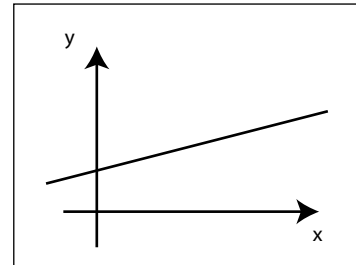


Figure 6-10. A simple line on a graph

To get the slope, or gradient, we just divide the height of the triangle by the width (a over b). Our triangle is of an arbitrary size, because technically speaking, our line is infinitely long, so we have to pick just a portion of the line to measure. The line in the far-right image actually has a negative gradient, because it is sloping downward. You can test this by dividing, say, -1 (the triangle's height) by 4 (the triangle's width) to get -0.25 .

The final part of the equation is c , the intercept along the y axis. That may sound complicated, but it simply refers to the point at which the line passes through the y axis. Therefore, a value of 0 for c , would mean the line passes through the origin. Figure 6-12 ties together all of these ideas.

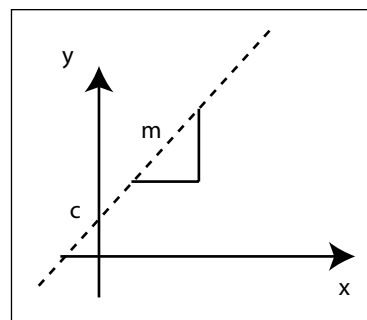


Figure 6-12. Triangle showing gradient (m) and intercept (c)

With that piece of rather dry theory out of the way, it's time to see how we can use this to actually detect a collision in our games.

Line intersection

This section relates to detecting whether a circle is touching a line—well, that's only half the story. In reality, we need to think of the circle as traveling along *another* line, a line that will, at some point, pass through the line that is our wall or obstacle, that is, if we extend the two infinitely. This second line is the circle's **path**. It's not guaranteed that the path our circle is traveling along *will* intersect the other line, however, as the lines might run directly parallel to each other. But for the moment, let's work on the premise that the lines are *not* parallel, and at some point along these infinite lines, they will cross and form an intersection point (see Figure 6-13).

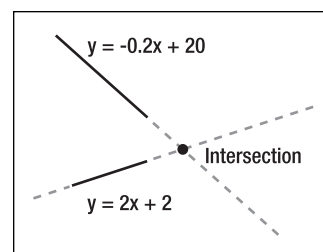


Figure 6-13. Lines intersecting at a point

For those wishing to dig into trigonometry a little more, just as you can find out whether two lines are parallel by seeing if their gradients are equal, you can also find out if they are perpendicular by multiplying the gradients together. If the result is -1 , your lines are at right angles to each other!

The intersection is the key, at this single point the x and y values in both line equations will be the same. This is the point we need to find. In Figure 6-13, I have labeled both lines with their corresponding equations:

$$y = -0.2x + 20 \quad (\text{for the wall})$$

$$y = 2x + 2 \quad (\text{for the circle's path})$$

If we go back to our general equation, we can think of these lines as follows:

$$y = m_1x + c_1 \quad (\text{for the wall})$$

$$y = m_2x + c_2 \quad (\text{for the circle's path})$$

Now that we have some pure algebra to play with, all we have to do is rearrange these equations to give us a value for x :

$$x = (c2 - c1) / (m1 - m2)$$

So, for our lines in Figure 6-13, this would be

$$x = (2 - 20) / (2 + 0.2)$$

which gives us $x = -8.19$ (approximately). Plugging the value for x back into either one of our original line equations we get the following:

$$y = -14.37 \text{ (approximately)}$$

I've rounded these numbers to save ink, but nevertheless, we now know that our two lines cross at the point with the coordinates $(-8.19, -14.37)$. With that in mind, now we only need to figure out if our circle is closer to that point than the length of its radius, just as we did when dealing with our circle collisions earlier on. As a result, this code may look a little familiar:

```
xDist = circleX - x;
yDist = circleY - y;
totalDist = Math.sqrt(xDist*xDist + yDist*yDist);
isColliding = (totalDist <= circleRadius); // true or false
```

If `isColliding` is true, we know that our circle's perimeter is breaching the line we are detecting against, so you can act on that in a suitable manner; for example, bounce the circle off the line as if it were a ball on a wall. We will look at doing just that shortly.

The line equations

There's one thing missing from our method; just how do we obtain these line equations in the first place? Not just guesswork or a protractor and ruler combination, that's for sure. Let's backtrack for a second, and look at how we can work out line equations. At this point, you might find it helpful to open `LineCircle.fla` from the Chapter 6 downloads (see Figure 6-14).

For the moment, let's not concern ourselves with the line equation for the path that the circle moves along. Instead, let's look at our wall's line; this solitary line is nothing more than a black stroke within a movie clip. What is important, however, is the location of this line in relation to the movie clip's registration point. Double-click the `line_mc` symbol *in the library* to open it for editing. The left-hand point of the line is exactly on top of the registration point, and the line extends 100 pixels to the right. This positioning gives us something concrete to work with in the next stage of the process, working out the equation.

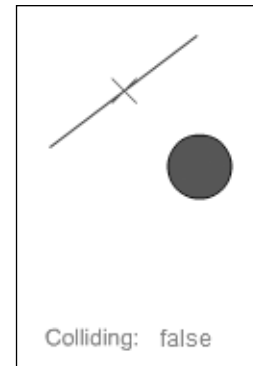


Figure 6-14. The LineCircle collision test example

Go to the main stage, and you can see I've rotated the line movie clip counterclockwise; to be precise, its `_rotation` property now reads around `-37.24786377` at runtime (you can check this in the Transform palette, available by pressing `Ctrl+T` or `Cmd+T`). Armed with the knowledge that Flash uses values ranging from `-180` to `180` for a movie clip's `_rotation` property, we can use this very angle to work out the gradient of the line; the code is very simple:

```
lineRot = line_mc._rotation * Math.PI/180;
line_x1 = line_mc._x;
line_y1 = line_mc._y;
line_x2 = lineLen * Math.cos(lineRot) + line_x1;
line_y2 = lineLen * Math.sin(lineRot) + line_y1;
line_m = (line_y2-line_y1)/(line_x2-line_x1);
```

Where `lineRot` is the rotation of the line in *radians*, `line_x1` and `line_y1` mark the registration point in the line (the start of the line); `line_x2` and `line_y2` mark the end point of the line; and `line_m` is the gradient that we get as a result. Again, we need to convert the degrees into radians, because they are much more natural for trigonometric functions, and therefore, radians are what `Math.cos()` and `Math.sin()` expect!

Now we have the target line's equation, but our work isn't quite finished. If you think back to the start of this section, I said you can define a straight line by the equation $y = mx + c$. We have m , but we need c , the y axis intercept. To get c , we need only the following equation:

```
line_c = line_y1 - line_m*line_x1;
```

That's it! We take the line's gradient multiplied by the starting x coordinate and subtract that from the starting y coordinate. I don't blame you if you are thinking, "Great. We have a bunch of numbers." But it will all make sense soon. Next, we need to follow the same process for the circle itself:

```
circRot = Math.atan2(yVel, xVel);

circ_x1 = circle_mc._x;
circ_y1 = circle_mc._y;
circ_x2 = circRad * Math.cos(circRot) + circ_x1;
circ_y2 = circRad * Math.sin(circRot) + circ_y1;

circ_m = (circ_y2-circ_y1)/(circ_x2-circ_x1);
circ_c = circ_y2 - circ_m*circ_x2;
```

The only difference here is that, in the first line, we use `Math.atan2()`. Because we don't have a line as such, we need to derive our imaginary line/path from the movement of the circle, in this case its x and y velocities. `Math.atan2()` will take our two figures, the values for y and x , and it will give us the angle the circle is headed!

Remember, if line_m, the line's gradient, is equal to circ_m, the circle's path-line gradient, you know the lines are parallel and will never cross, so beware—put a check in to make sure you don't continue with the detection if these lines are parallel.

Putting it all together

We now have the figures required to build the line equations for the wall and the path the circle is traveling along. Let's plug those values into our line intersection algorithm and see what we get:

```
intersectX = (circ_c-line_c)/(line_m-circ_m);
intersectY = line_m*intersectX + line_c;
xDist = Math.abs(circ_x1 - intersectX);
yDist = Math.abs(circ_y1 - intersectY);
totalDist = Math.sqrt(xDist*xDist + yDist*yDist);
```

Well, there we have it, a value for intersectX, intersectY, and for good measure, the total distance between the center of the circle and the point on the line where that intersection occurs (totalDistance). To clearly mark that point, I'm moving a crosshair to the intercept position in the example with the following code:

```
intercept_mc._x = intersectX;
intercept_mc._y = intersectY;
```

This gives us a visual representation on the point that we've worked out for the circle to cross the wall line. The final step is to check whether the point at which the circle is supposed to collide with the line is actually on the line. Remember that we extended the line infinitely in the math, so we need to rein that back in to see whether the intersection point is along the stretch that is displayed on the screen:

```
if ( (intersectX >= line_x1 && intersectX <= line_x2)
    && (intersectY <= line_y1 && intersectY >= line_y2)

    || (intersectX >= line_x1 && intersectX <= line_x2)
    && (intersectY >= line_y1 && intersectY <= line_y2)

    || (intersectX <= line_x1 && intersectX >= line_x2)
    && (intersectY <= line_y1 && intersectY >= line_y2)

    || (intersectX <= line_x1 && intersectX >= line_x2)
    && (intersectY >= line_y1 && intersectY <= line_y2) )
{
    // Now for the old circle radius distance test
    isColliding = (totalDist<=circRad);
}
```

I think you'll agree that is a pretty verbose if statement in Flash Lite 1.1 syntax, but it works and runs fairly quickly.

Collision reactions

So far, all of the talk regarding collisions has been about the detection. This is great if your objects are programmed to vaporize or explode on contact, but you might want to model other reactions, such as an object bouncing off of a surface or the collisions several objects affecting each others' directions and velocities. One statement you may remember from school is

$$\text{Angle of Incidence} = \text{Angle of Reflection}$$

Figure 6-15 illustrates this with a ball bouncing against a wall.

Of course, this doesn't take into account gravity or other forces, but another example might be a pool ball on a pool table hitting the cushion and rebounding. The thing to note is that the reaction is equal and opposite. When we don't take into account things like friction, what we have is an elastic collision.

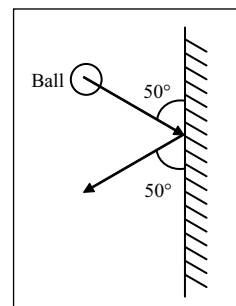


Figure 6-15. Ball hitting a wall and rebounding

Elastic collisions are collisions where no momentum is lost. As you can imagine, this means we have less to compute in the game, making it a little easier on the processor. If we wanted to model the real world more accurately we would have to make all collisions inelastic, energy is always lost in a collision between objects of size;¹ the lost energy is converted to another form such as heat or light. But in the interests of fun, we are not going to concern ourselves with such details.

The final part of our line-to-circle collision detection example actually shows a little more code—the code that makes the circle (or ball) bounce off the line (or wall). Here is that code:

```
if (isColliding)
{
    // grab the angle for the line's tangent/gradient
    alpha = Math.atan(line_m);

    // apply velocities to gradient/tangent line
    yVel2 = yVel * Math.cos(alpha) - xVel * Math.sin(alpha);
    xVel2 = xVel * Math.cos(alpha) + yVel * Math.sin(alpha);

    yVel2 = -yVel2; // flip vel along ball path

    // re-apply velocities to original trajectory/path line
    xVel = xVel2 * Math.cos(alpha) - yVel2 * Math.sin(alpha);
    yVel = yVel2 * Math.cos(alpha) + xVel2 * Math.sin(alpha);
}
```

1. By “collisions of size,” I mean the macroscopic collisions we are modeling in our games between cars, missiles, billiard balls, and such. We aren't concerned with the strange goings-on that can occur at the microscopic level.

Thankfully, there's not too much here. Essentially, what we are doing is what the diagram at the start of this section showed—making the ball bounce *back* at the opposite angle at which it struck while keeping its momentum. The first line of code here gets an angle, `alpha`. `alpha` is the angle of the line's gradient (or tangent, with it being a straight line and not a curve). We get this using `Math.atan()`; the result is in radians.

We next create a couple of temporary x and y velocities, `xVel2` and `yVel2`. These are made by plugging the angle `alpha` into two sets of equations. The first multiplies the existing velocity by the cosine of the angle, subtracting the same value by the sine of `alpha`. The second performs a similar operation, but it *adds* the sine instead. If this doesn't make sense, do not worry; I had to go back to my geometry books before I remembered why we do this. Simply put, this works out the velocity of the ball in relation to the line, you might say it was “mapping” the x and y velocity onto the equation of the wall line.

The next line flips the y velocity, `yVel2`. We do this so that the ball actually bounces off of the wall and doesn't just pass through it! This bit technically does the real work we set out to do. The final couple of lines take our new x and y velocities and map them back onto the original path line, so that we can apply our new `xVel` and `yVel` to the ball to see the reaction take place, and voilà, a collision reaction.

If you would like to sap some of the momentum out of the ball when it hits the wall, just multiply `xVel` and `yVel` by a number less than 1, like 0.5. On top of that, you might be applying a friction value to these two figures every frame that naturally reduces the velocity, like we had with gravity in the jumping sheep example when dealing with the jump velocity.

Why so slow?

If you've taken a moment to test the previous sample on your device, you will notice that it runs at a leisurely pace. The phone's CPU is just not up to the challenge yet. In the next section, we will make a little headway with optimizations, but I have included the previous examples regardless of speed, because there is no doubt at all about the increasing speed of handsets. Personal tests have shown that the speed of Flash Lite-capable phones has doubled in the last six months, so this code will be useful in the long run when processor speed is less of an issue. This applies even more so for PDA devices, which are already fast enough to run the previous example and already support Flash Lite 2. Detecting collisions against just one or two lines as opposed to five (for example, in something like Pong) is another example of where this is still useful without hampering performance. One thing using only two lines doesn't solve, however, is the need to figure out all of the properties of these lines at runtime, so that we can plug those values into the equations.

One immediate solution is to realize that the lines are never moving, so we can simply hard code the values for their rotation, their length, and any other information we might need. In fact, we can perform what is known as precalculation, and I'd like to cover that in the next section.

Efficient math

We've already looked at a few things that you can do with code to speed up the performance of your applications. One example is to use circle-to-circle collision detection between a player and another sprite or two vehicles. This is far more efficient than using the line math or even bounding-box collision approach, which technically could be more accurate. Another technique we can use that used to be the norm back in the Flash 4 days is something called precalculation.

Precalculation

Precalculation is the process of working out, ahead of time, all the possible values an expression can generate. To put it another way, if we want to convert an angle in degrees to radians, using $\pi/180$, to the angle in radians, we could also generate an array of 360 values one time only before we begin running our game or application. This means less accuracy, of course; with 360 values we have only enough for one per degree, so any values in between would need to be rounded up or down to a whole degree. However, precalculating gives us an *accurate enough* set of numbers to use without needing to perform the multiplication perhaps several times every second. This works in any case where a simple array lookup (using the array access operators or `eval()`) is faster than evaluating an expression. Let's run through this example first in Flash Lite 2 code:

```
var radians = [];
for( var i=0; i<360; i++ )
    radians[i] = i*Math.PI/180;
```

Later on, we might use this like so:

```
var valueInRadians = radians[ int(degreeValue) ];
```

This is not a great example of using precalculation to good effect, because the equation we are calculating a table for is very simple. But in our circle-to-line collision example, we can make good use of pregenerating values for later use.

If you remember our circle to line collision example was pretty slow going. So let's take a look at `WallsCircleOptimized fla`, where I've gone through and performed a bit of optimization. The first thing you might notice is the hard coding of a couple of values:

```
lineLen = 100;
circRad = circle_mc._width/2;
d2r = Math.PI/180;
```

By hard coding these values, we reduce the number of instructions the Flash Player needs to execute in the loop. Things like `Math.PI/180` are never going to change, so make use of this wherever possible.

Next, we precalculate some information about the lines:

```
for (i=1;i<=5;i++)
{
    lineRot = eval("r_" add i) ↵
              = eval("line" add i add "_mc")._rotation * d2r;

    line_x1 = eval("x1_" add i) ↵
              = eval("line" add i add "_mc")._x;

    line_y1 = eval("y1_" add i) ↵
              = eval("line" add i add "_mc")._y;

    line_x2 = eval("x2_" add i) ↵
              = lineLen * Math.cos(lineRot) + line_x1;

    line_y2 = eval("y2_" add i) ↵
              = lineLen * Math.sin(lineRot) + line_y1;
```

```

        line_m = eval("m_" + i) ↵
                = (line_y2-line_y1)/(line_x2-line_x1);

        line_c = eval("c_" + i) ↵
                = line_y1 - line_m*line_x1;
    }

```

This might look a little convoluted, but it is essentially fairly simple. What we have here is a loop that runs through each of our five lines, using the `eval()` statement to create short variable names like `r_1`, `r_2` and so on and assigning values like the rotation of each of the lines. Next, we assign that rotation value to another local variable for use later in the loop. What we end up with is perhaps 35 new variables (40 if you include the local ones) that can be accessed in our collision detection, without having to constantly recalculate all of these rotations, gradients, and so on, which are fixed and never change in this particular case. This has a dramatic impact on how fast the code executes. The balance really is to keep it readable without sacrificing too much speed. Test the sample on a phone to see the difference; it runs at approximately twice the speed!

Profiling

On the desktop, I find it useful to use what is known as a **profiler** to check how long my programs are spending in the various functions that comprise them. A profiler can be started and stopped wherever required and lists the number of milliseconds each function takes. A very simple example of this follows:

```

    startTime = getTimer();
    // profiled code here
    for( var i=0; i<1000; i++ ) { trace(i); }
    // profiled code ends
    trace(getTimer() - startTime);

```

What we have here is the number of milliseconds our simple for loop took to execute. An alternative is to use ASProf by David Chang, which is written for ActionScript 1.0 but also works for ActionScript 2.0 in Flash Lite. You can download ASProf as an extension for Flash from www.nochump.com/asprof.

To use ASProf, you simply include the ASProf ActionScript include file, which is automatically transferred to the correct place, when you install the extension and specify which classes or methods you wish to profile. This works for ActionScript 1 prototypes or ActionScript 2 classes:

```

#include "ASProf.as"

import com.test.MyClass;

ASProf.profileObject("com.test.MyClass.prototype");

ASProf.begin("a test");
tester = new MyClass();
tester.test();
tester.test2();
ASProf.end();

trace( ASProf.getFlatGraph() );

```

In this example, we're creating a new instance of the `MyClass` class and setting the profiler to examine its prototype. For those of you who have never known the delights of working with the prototype object, this is how classes in ActionScript 1 and (behind the scenes) in ActionScript 2, are actually achieved. Telling `ASProf` to profile the prototype of a class means that it will examine all functions you call on it. Alternatively, you can just specify one method and examine that. You can run this example yourself from the `Profiling.fla` FLA in the `Profiling` folder of the Chapter 6 downloads. The `test()` and `test2()` methods don't do much, but when you call `ASProf.getFlatGraph()`, you get a nice table showing the exact number of seconds spent in each function. You can also tell that `test2()` is taking 98 percent of the total time. It is pretty obvious why when you look at the actual functions (`test2()` has a for loop that goes around 99,999 times!); nevertheless, in the wild, this is a great way to find bottlenecks and optimize code.

Using short variable names

This tip is going to sound a little bit strange, but the age-old practice of short variable names really does have a noticeable impact on code that is run in Flash Lite, and you even save a bit on file size. We use this exact technique in the previous example of the optimized wall (circle-to-line) collision test. I know it can be a lot of effort, but if you are using external ActionScript files and a good editor like Eclipse with the ActionScript Development Tool (ASDT), which is available for free at www.osflash.org/asdt or FDT, which is available commercially at <http://fdt.powerflasher.com/flashsite/flash.htm>, you can quite easily find and replace multiple files to take out a lot of the effort. This technique is best used in small blocks of code that you have determined are processor hogs by profiling, rather than applying this across your whole application.

Game assets

Assets are any bitmap images, sounds, videos, or vector graphics used in your games. If you were a J2ME developer, you'd probably be using mainly small bitmap images, with the occasional sound to build your games. Thankfully with Flash, we also get the option to quickly build scalable graphics using the tools available in the IDE. These sorts of graphics are best when kept simple; a lot of vector lines can really slow down your game (think back to our circle-to-line collision vector math earlier in this chapter). In this section, we will look at the various types of assets you can use in your games and how you can go about optimizing them.

Vector graphics

This is what made Flash famous at the start. Vector graphics are most commonly created in the Flash IDE or another program like Adobe Illustrator. All lines and fills used to make up an image are described mathematically behind the scenes; we just manipulate them using tools like the pen and fill bucket. This method of defining graphics means that they can scale to any size without losing clarity. They can give you better-looking graphics at virtually any resolution, and you can rotate these graphics without the bitmap pixilation and zigzags associated with rotated bitmaps—not to mention that the file size can be kept down, but this is something of a balance. When you use vectors, the odds are that you are not creating very detailed images, which means the file size will be less than the bitmap equivalent. But with smaller images and images that contain many vectors, sometimes vector drawings will result in larger files than their bitmap equivalents.

I work with designers on a daily basis, and there's a distinct difference between how most designers think about Flash when compared with coders. Many designers, for very good reasons, only care about the end result and how it looks. But it is our responsibility as developers to make sure we are making best use of the tools available. This means splitting up parts of a graphic into symbols (like movie clips), so that you can reuse or animate those symbols without having to create new graphics.

A real example of this is shown by the two files `ArmAnim.fla` and `ArmAnimOptimized.fla`. The first file shows an animation of an arm bending, done simply by drawing each position of the arm as it moves. That's quite a classic approach, but the file size is a whopping 3KB just for that one tiny piece of animation. Imagine an entire game's animation! The second file shows what would happen if we split the arm into three separate graphic symbols (see Figure 6-16). Not only is it quicker to make the animation, but the file is one-third of the size. We could also reuse these parts without adding much to the file size at all!

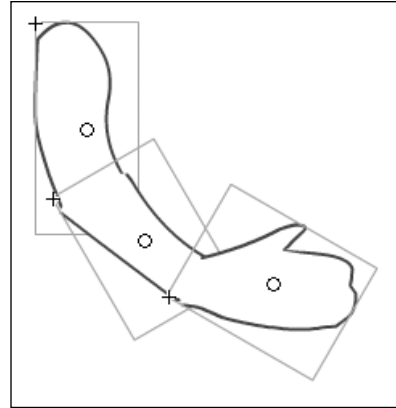


Figure 6-16. The arm animation broken up into graphic symbols

When it comes to optimizing our vectors, we can use one of two tools that Flash offers. The first is the smooth tool. Just select some vector graphics in the IDE, and press the S-shaped button with an arrow on it that appears at the bottom of the Tools palette to perform a simple smooth operation. Alternatively, press the button showing an angled line, which is to the right of the button you used to smooth, to do the opposite. Both operations usually result in fewer lines in your vector graphics, and thus, better-optimized graphics. Taking this one step further, you can also access the second tool available to us, the optimize operation, available from the `Modify > Shape > Optimize` menu shortcut. This actually sets about reducing the number of curves and lines used to make up your vector graphics. Take a moment to play around with the settings; you can adjust the slider to perform varying levels of simplification to your graphics until you are happy with the final result.

Another tip for avoiding sluggish animation is to stay clear of the `_alpha` property. Using `_alpha` is a sure-fire way to slow things down, especially when the graphics are complex. If at all possible, it is recommended that you alter the colors in your graphics to match your background instead of altering their opacity to allow the background to show through.

Bitmap graphics

Bitmap graphics can be imported into Flash in a couple of ways. You can use the `File > Import` menu item, and selecting your bitmap file, such as a PNG, BMP, or JPG. Or you can simply copy and paste the data from the clipboard in, say, Photoshop. I recommend the first way, so that you can make use of the update command. The update command is available by right-clicking any bitmap asset in the library and choosing update. This will automatically reimport a bitmap file if you have edited the file on the hard disk and automatically update any other library symbols that may use it.

With bitmaps, the rule is to keep them as small as you can. We already have a smaller screen, which helps, but keeping the sheer amount of bitmap data to a minimum is a good way to avoid memory

hogging. Remember that in order to display even a 1KB JPG, the Flash Player must completely decompress the data into red, green, and blue (RGB) pixels at runtime, so the actual amount of RAM used is something like this:

$$\text{RAM used in bytes} = \text{width} * \text{height} * 3$$

One thing we can do to optimize bitmaps is to split them up into tiles. This means we can reuse individual tiles, or sprites, in various places. But it also means that with larger bitmaps, we don't necessarily have to have all of it decompressed in memory at once.

I've included a file to show this. Open `pano.fla` from the `Panorama` folder. You can see that I've split up the wide image into four parts. Without splitting the image, it actually uses too much RAM for my Nokia to cope with. Splitting up the image allows the effect to be achieved, because we are only ever showing two of the bitmaps at once. One final tip that deserves mention is that it helps to import your graphics at the size you wish to use them instead of scaling them down. It's an extra step but well worth it ultimately.

Sounds and music

There are two main types of sound you can use in Flash Lite, device sounds (such as MIDI or MFi) and standard sounds (such as MP3). The major distinction to make here is that with Flash Lite 1.1, we must embed the sounds in the SWF. With Flash Lite 2.X, we can use the Sound object to dynamically load sounds from the phone or over HTTP.

Device sounds are formats that are native to a device like MIDI or MFi. These are not supported in the desktop player, but we can use them on a mobile device if it already supports them, as most do now for things like ring tones. As with video (see the example in the following section), you should check the `System.capabilities` object to see which types of device sound are supported. In Flash Lite, we must use proxy sounds to represent our device sounds. **Proxy sounds** are regular Flash audio files (like MP3 or WAV) that you include in your library as usual. When you right-click these and choose Properties, you will notice a few more options when publishing for Flash Lite, as shown in Figure 6-17.

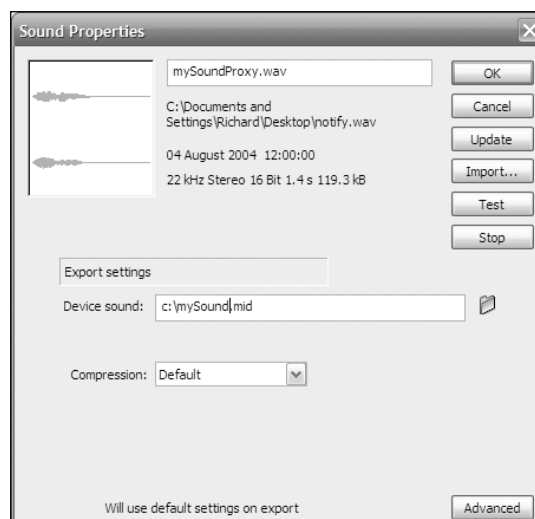


Figure 6-17. The Sound Properties panel for device sounds

The options in this panel let you link your MP3 or WAV file to a device sound, so that when the SWF is actually built, your native sound gets replaced with the device sound (usually meaning it has a smaller file size), and when it's played in the emulator or on the handset, you hear the right thing!

Standard Flash sounds, as I mentioned, are usually MP3 or WAV files. In the library, you can access the Properties dialog for these to choose the actual compression used when compiling your SWF, for example, ADPCM or RAW. These sorts of sounds are played back in a similar fashion to the way they are played back on the desktop player, usually by placing them on keyframes. We can also load them at runtime using the Sound object in Flash Lite 2, in particular, using the `Sound.Load()` function, which can be used to load a sound from a given location, and `Sound.start()`, which can play the loaded sound. The Flash documentation includes a help book on Flash Lite 2, which goes into details on using the Sound object. But you can also use them in the regular keyframe fashion by selecting any keyframe on the time line, and in the Properties panel, choosing an option from the Sound drop-down menu, as shown in Figure 6-18.

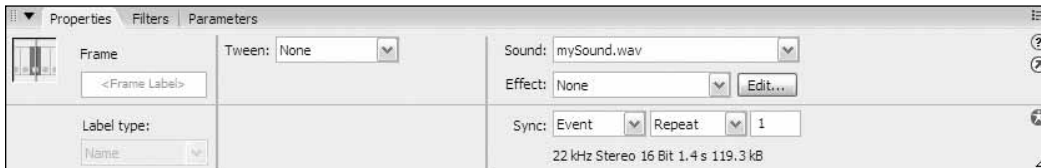


Figure 6-18. The Sound options drop-down menu

The Sound drop-down menu also gives you options to repeat the sound, for example, with a background loop. My biggest tip regarding sounds in games is that you give the user an option to turn them off. Perhaps make use of an MSO (described in the next section) to save that particular preference.

Video

There are three main ways to play video files on a mobile phone using Flash Lite. The first is really a hack—you can simply fill a time line with a series of PNGs or JPGs and play back that animation as normal. This method is required for Flash Lite 1.1, but it is not very efficient and results in some large files, so if possible, it is to be avoided.

In terms of actual video files, we can play video using 3GP and MP4 files. I've included a sample file that shows a video jukebox application capable of playing a playlist of video files included with the SWF. You can look at this example in `VideoJukeBox.fla` in the Chapter 6 downloads (see Figure 6-19). Unfortunately, legal restrictions prevent me from including the video files themselves, but you can replace the filenames with any 3GP or MP4 files you can download freely on the Net, and the example will work with those too.

Unlike with SWF files prepared for the desktop Flash Player, we cannot embed Flash video in the FLA directly. The Flash Lite Player cannot render FLV video. But Adobe has made things easy for us. Just create a blank video symbol in the library using the drop-down menu at the top-right corner of the Library panel, drop it onto the



Figure 6-19. The minitoobr video jukebox application

stage, give it a name (`myVideoInstance`), and then simply use `myVideoInstance.play(filename)` to actually begin rendering the video to screen. It's much easier than messing with `NetConnection` and `NetStream`, as we have to with desktop Flash!

In terms of size and compression ratios, I'd recommend keeping video to a standard known as Quarter Common Intermediate Format (QCIF), which is 176×144 pixels. There is a smaller standard size of Sub QCIF (128×96 pixels) that you should try if you would like to support older handsets, but with the standard QCIF, you can pretty much guarantee problem-free rendering on modern handsets. There are many tools that can compress files for you. One is the Nokia Multimedia Converter, downloadable from <http://forum.nokia.com>. Other tools include FFmpeg (<http://sourceforge.net/projects/ffmpeg>), which is a free and open source video conversion tool.

Saving and loading high scores

We can make use of MSOs to load and save data from the phone's memory. I think the best way to show the use of MSOs is with an example. This example will demonstrate adding a new high score, saving all the high scores, and loading them back in when the application restarts.

On stage, we need four things: a large dynamic text field named `output_txt`, and three simple buttons named `add_btn`, `save_btn`, and `clear_btn`. These can look however you wish, or you could just open `Highscores fla` from the downloads and dig right in. Now for some code, let's begin on frame 1 as usual:

```
// Set up movie
fscommand2("fullscreen", true);
var scores:Array = new Array();
```

There's not too much to take in so far; the only thing of interest here is that we create a new `Array` to store our high scores in. Next, we have the code required to set up our MSO and load it:

```
// Load in MSO
SharedObject.addListener("scoresSO", this, "onLoadMSO");
var so:SharedObject = SharedObject.getLocal("scoresSO");

// Called when MSO loads
function onLoadMSO(so:SharedObject)
{
    output_txt.text = "Scores loaded:\n";

    // If this is not the first time
    if ( 0 != so.getSize() )
    {
        scores = so.data.scores;
        displayScores();
    }

    SharedObject.removeListener("scoresSO");
}
```

The first line adds a listener to the `SharedObject` class. This is different from the desktop version, because shared objects are not instantly accessible in Flash Lite. We also define the scope and the function name to call when the `scoresSO` shared object is loaded. The next line begins loading the shared object with a call to `SharedObject.getLocal()`. Finally, the function `onLoadMSO` is called when our MSO is available; inside this function, we check the size of the `SharedObject` being loaded. If it is 0, we know it is the first time around, so we do not interrogate its `.data` property for our high scores array. However, if it is larger than 0 bytes, we know to grab a copy of the `so.data.scores` array and call the `displayScores()` function to loop through it and output the values to our text field. The final line in this function just removes the listener from the `scoresSO` shared object. This is not really needed, but it is always good practice to clean up after yourself when you know you won't be loading it again in the application's lifetime. This applies to any time where you find yourself adding event listeners to objects.

Next, we have our button-click handlers, starting with `add_btn`:

```
add_btn.onRelease = function()
{
    scores.push( {name:"name"+random(9),
                 score:random(999)} );

    displayScores();
}
```

Having already created a blank scores array (or loaded one in from an MSO), we simply append to it a new random name and score wrapped up in an anonymous simple object with `.name` and `.score` properties, respectively, and call `displayScores()` just so that the user knows what we added.

Next, we add the functionality for `save_btn`:

```
save_btn.onRelease = function()
{
    so.data.scores = scores;
    so.flush();
    output_txt.text = "Please close and restart app.";
}
```

The Save button's job is to assign our (possibly) modified scores array to the `.data` property of our MSO and then call `SharedObject.flush()`, which writes the data to the phone. Giving the user feedback on the next line is always good practice.

Now, let's look at `clear_btn`:

```
clear_btn.onRelease = function()
{
    so.clear();
    output_txt.text = "Scores cleared.";
}
```

I put this button in just to show that you can always call `SharedObject.clear()` to delete a `SharedObject` from the phone's memory. Finally, we have the `displayScores()` function we called a few times previously:

```

function displayScores()
{
    output_txt.text = "";

    for( var i=0; i<scores.length; i++ )
    {
        output_txt.text += scores[i].name;
        output_txt.text += "\t\t";
        output_txt.text += scores[i].score;
        output_txt.text += "\n";
    }
}

```

In this function, we simply loop through the scores array and append the name and score values to our output text field. Note the use of "\t" and "\n" to insert tab stops and new lines, respectively.

That's all there is to it. The ability to save a Flash native Array without having to parse it or convert it to a string and save it on the server is a massive bonus to Flash Lite development. Remember that you can add almost anything to the .data object of the SharedObject class, including maybe the total playing time in minutes or the number of times the player has opened the game!

Sample games

We've had a few small examples along the way in this chapter, but I thought it would be a good idea to include some complete examples of games for you to dissect and adapt at your leisure. Hopefully, you can spot all of the techniques introduced in this chapter in these games. Feel free to customize them or introduce new elements as you wish. I'd recommend reading through this chapter again once you've had a chance to look at these games, just to help build a good solid foundation to work from.

Mad Bomber (Flash Lite 1.1)

Mad Bomber is a version of a classic arcade game called Kaboom!, originally made for the Atari 2600. Not only is this game addictive but it's easy to make. The concept is simple: a deranged maniac is dropping bombs from the top of a wall. You, the hero, must stop the bombs from reaching the ground by catching them in your bucket of water, of course.

You can find Mad Bomber (as well as all of the other games) in the Chapter 6 downloads. The controls to this game are very straightforward—just a left and right arrow system and simple menu buttons for further interaction (see Figure 6-20). This should remind you of some of the key catcher code from previous chapters.



Figure 6-20. Mad Bomber

Snake (Flash Lite 2.0)

The second game for your perusal is based on the famous game that came with the earlier Nokia mobile phones, Snake (originally called Nibbles!). Snake, shown in Figure 6-21, leverages the Flash Lite 2 pure-code approach described in Chapter 3. The use of ActionScript 2.0 makes it very easy to develop and reuse this code for other games. You can find the source in the `ch6/Snake` folder. Each of the class files is fully commented.

The game consists of the following four classes:

- SnakeGame: Controls the game
- SnakeWorld: Draws the world of the game
- Snake: Draws the snake on the screen
- FoodPiece: Draws food pieces on the screen

Let's take a brief look at the classes; you may wish to have them open in Flash while reading this, so that it is easier to follow along.

SnakeGame

The SnakeGame class is the daddy, responsible for controlling the interaction in the game. It keeps track of the score and overall state of play, and it creates an instance of the SnakeWorld class. The communication between the SnakeGame and SnakeWorld classes is handled via events using the EventDispatcher model.

SnakeWorld

The SnakeWorld class is a visual class used to draw all the graphics of the game, such as the background, the food pieces, and the snake itself. The SnakeWorld class also receives events from the Snake instance that it creates, in order to know when the snake eats a food piece, but also when it hits the sides of the screen; or, of course, itself. The SnakeWorld class also tells the Snake instance to update its physical position every frame.

Snake

The Snake class is the most important class of the game, perhaps. It includes most of the logic found in the game, for sure. Its main job is to draw the snake graphics, though, and make them move. When the Snake class gets created, the head of the Snake is added to an array of items that need to be drawn on the stage. The items in this array make up the segments in the snake. Each time the snake has to update itself, the following steps are executed:

1. The positions of the segments of the snake get shifted forward.
2. The position of the snake head is updated based on the current direction.
3. The snake is checked to see if it has collided with the sides of the screen.
4. The snake is checked to see if it has collided with a food piece, itself, or its tail.

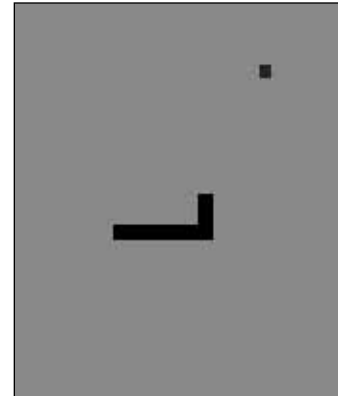


Figure 6-21. Snake

The snake moves by looping through its segments' array, shifting each part of the body into the position of the next, right up to the head before finally moving the head to a brand new location, depending on which direction it is heading in. If any of the collisions in step 3 or step 4 occur, an event will be dispatched to give a signal to SnakeWorld that something has happened. If the snake eats or collides with a food piece, a FoodPiece event will be dispatched; each time this event gets caught by the SnakeWorld class, a new FoodPiece will be drawn on the screen.

FoodPiece

The FoodPiece class is a simple class attached to a MovieClip symbol. It is used only to draw the food graphic on stage, using the attachMovie() function and the linkage ID specified by the LINKAGE variable in the class. Using something like a LINKAGE static string for each class linked to a library symbol is good practice, because it allows for automatic checking of the first parameter when using attachMovie() (meaning if you mistyped SomeClass.LINKAGE, you get a compile error, whereas if you mistype a plain linkage ID string, you merely don't get anything on stage).

BlackJack (Flash Lite 2.1)

The BlackJack example, shown in Figure 6-22, is perhaps the most complicated example in this chapter. It makes use of the Flash Lite 2.1 XMLSocket connection to connect to a Java server that could be running online anywhere. The Java server acts as the dealer in a game of blackjack. This example is a very simple form of the game, but you could use this to develop a very polished version complete with nice graphics for the cards (instead of just numeric values) as well as animations for going bust or winning.

The game works by establishing a connection with the server using XMLSocket. Then, messages are sent as XML objects. The server reads these as plain text, but it could just as easily parse these into, say, a Java XML class, so that it can read the individual nodes. In our case, the XML messages are simple enough for us not to need this, but it is always there just in case. The server then responds with more XML messages (constructed and sent as strings), which the Flash client can parse and react to. There's not too much code on the client; you can find it all in two frames on the main time line.

To start up the Java server make sure you have a Java Development Kit (JDK) installed; you can get the latest, JDK 5, at <http://java.sun.com/javase/downloads/index.jsp>.

I recommend the version with the NetBeans IDE, if you don't have an editor already, unless you prefer to use an IDE like Eclipse to edit your Java code, as I do. Either way, you can use the source provided without any other tools. First, you need to add the JDK bin folder to your systems PATH or Environment variables, so that you can run java.exe (the Java virtual machine) and javac.exe (the Java compiler) from any command-line window and any folder. If you don't know how to do this, please search online for "setting path in Windows", or "setting environment variables Mac", depending on which system you use, for several useful articles on the subject. You can test whether the JDK is all set up by opening a command prompt, navigating to the folder containing the sample .java files included, and typing

```
java -version
```



Figure 6-22. BlackJack

If you see something like `java version 1.5.0`, you know you are on track. With that done, you need to compile the `.java` files into Java byte code (`.class`) files, so that we can run them in the Java virtual machine. To do this, type the following:

```
javac BlackJackServer.java
```

I've also included a batch file to run this for you (`compile_server.bat`). This file generates the class files, ready to run. To run them, type the following:

```
java BlackJackServer
```

You should see `BlackJackServer` running on port: 2048. Alternatively, I've included another batch file to do this for you. You can terminate the server at any time using `Ctrl+C` or closing the Command window. You are now ready to compile and run the FLA! Be sure to check out the Command window; it will show the messages coming through.

Summary

In this chapter, we have explored a whole range of game-related topics. There is enough in the subject of Flash game programming alone to fill several large books, so hopefully, this chapter gives you a good foundation to get started making some games for yourself, having covered getting your game assets together, optimizing coding, building in logic and physics, and more.

In the next chapter, we will look at mobile content such as screensavers and wallpapers, another area where Flash Lite can be used.

