Chapter 2

Selected ActionScript 3.0 Topics

ActionScript 3.0 is very different from the previous versions of the language. This chapter will give a brief review of the elements of AS3.0 that we will make most use of in the rest of this book. It is not meant to be a comprehensive tutorial on ActionScript; instead, it is a summary of what you need to know to understand the code examples in the book. The other aim of this chapter is to cover relevant aspects of Flash and ActionScript that will set the context for applying physics.

This chapter was written with the assumption that the reader would have at least a basic knowledge of Flash and ActionScript. If you are an experienced AS3.0 programmer, you could safely skip most of this chapter, perhaps skimming over some of the material at the end on ActionScripted animation and collision detection. On the other hand, if you haven’t done any programming with ActionScript before, we suggest you pick up one of the books mentioned in the summary at the end. If you have programmed in an earlier version of ActionScript, or maybe need a refresher of AS3.0, you will benefit from going through the chapter in some detail. While the overview on its own won’t make you a proficient AS3.0 programmer, it should enable you to use and build upon the code examples in the book without much difficulty.

Topics covered in this chapter include the following:

- **ActionScript 3.0 classes**: Classes and objects are the basic building blocks of object-oriented programming (OOP). “Things” in the real world are represented as objects in ActionScript. Objects have properties. They can also do things by using methods.

- **ActionScript 3.0 language basics**: For completeness, the basic constructs of AS3.0 and their syntax are reviewed, such as variables, data types, arrays, operators, functions, math, logic, and loops.
• **Events in ActionScript 3.0:** The event framework in AS3.0 is very different from what it was in previous ActionScript versions. We briefly review some basic concepts and syntax, giving examples of how to make things happen in response to changes in the program or user interaction.

• **The Flash coordinate system:** This is the equivalent of space in the Flash world. Objects can be positioned on the stage in 2D and 3D using code. We review the differences between the Flash coordinate system and the usual Cartesian coordinate system in math.

• **The Flash drawing API:** The ability to draw things using only code is a powerful tool, especially when combined with math and physics. Some of the most common methods of the AS3.0 drawing API, which will be used throughout the book, are briefly reviewed here.

• **Producing animation using ActionScript:** We review different methods of producing animation using code, and explain the main method we’ll use for physics-based animation in the rest of the book.

• **Collision detection:** In any game or other project involving motion, things are inevitably going to bump into each other. Collision detection algorithms tell you when this has happened so that you can program the appropriate response.

## ActionScript 3.0 classes

If you’ve programmed using an earlier version of ActionScript before version 3.0, chances are you haven’t worried too much about classes. However, in AS3.0 it is impossible to ignore classes, because they are built into the very structure of the language. So what is a class?

A **class** is a fundamental construct of **object-oriented programming (OOP).** It is an abstract specification of an object that you want to create. An **object** is anything that exists in your movie at runtime.

Now that sounds pretty abstract, so let’s look at an example. Suppose you want to create particles in a project, and the particles will be the objects. How will Flash know how to make particles? Well, you’ll first have to tell it what a particle consists of, how it should look, what it can do, and so on. This is done by creating a class for the particle object called Particle, for example (by convention, class names start with an uppercase letter). Then each time you want a particle, you invoke the Particle class. The following sections describe how to produce an object from a class, create a class, and understand what it consists of.

### Classes and objects

Let’s take another look at the BouncingBall class from the last chapter. The function init() contains an example of how to produce an object from a class:

```actionscript
private function init():void {
    ball = new Ball();
    ball.x = 50;
    ball.y = 75;
}```
addChild(ball);
    addEventListener(Event.ENTER_FRAME, onEachTimestep);
}

The ball object is an *instance* of the Ball class, and is created (or *instantiated*) by this code:

    ball = new Ball();

The ball is subsequently made visible (technically added to the *Display List*) by the addChild command:

    addChild(ball);

Note that the Flash movie is made from a class. (See the listing in Chapter 1 for an example.) In Flash CS3 to CS5, this is called the *document class*. In Flex Builder 3 and Flash Builder 4 (used for a pure ActionScript project), it is generally called the *main application class* or something similar.

**Structure of an AS3.0 class**

Okay, you now know how to produce an object from a class, but what does a class actually look like in AS3.0? Here is a simple example of a perfectly valid class that does nothing:

```plaintext
package{
    public class DoNothing {
        public function DoNothing() {
        }
    }
}
```

Note that the class specification is enclosed within a `package{}` statement. A *package* is a bit like a folder; it is used to organize related classes. The default package has no name, as in this example. But you could give it any name you want, for example `package mypackage{}`.

The class is then declared using `public class DoNothing{}`, where `DoNothing` is the name of the class in this example.

This class ships with a *function* (more precisely a *method*) with the same name as that of the class: the `DoNothing()` function. This function is called the *constructor* of the class.

The keyword `public` sets the scope of a class, function, or property (we’ll introduce properties shortly). This means that the class `DoNothing` and function `DoNothing` can be accessed from outside the class. It is also possible for a class to have *private* functions, which can be accessed only from within the class; or *protected* functions, which can also be accessed by its subclasses (see the following section on inheritance). We’ll show examples of private and protected functions soon.

The next few sections explain some of these concepts in a bit more detail.
Functions, methods and constructors

As you've just seen in the last subsection, an example of a function is the constructor of a class. The code within the constructor is automatically executed whenever the class is instantiated as an object using the new keyword.

In the BouncingBall example, the function init() is called from within the constructor. Note that init() has private scope, so it is accessible (and therefore callable) only within the class BouncingBall. Why do we need a separate function init() in this example? Couldn't the code in init() be placed directly inside the constructor? The answer is that code placed in the constructor is interpreted, whereas code placed anywhere else is compiled—and so runs faster.

Notice the use of the function type void (this was Void, with a capital V in AS2.0). This means that this function does not return any value. It is possible for functions to return different kinds of things; for example, this function returns a random number between 0 and 100:

```javascript
private function randomNumber():Number {
    return(Math.random()*100);
}
```

As in AS2.0, functions (and, therefore, methods and constructors) can have arguments or parameters (values that we can feed into them). For example, this function adds two numbers supplied as arguments and returns the value as another number:

```javascript
private function addNumbers(x:Number,y:Number):Number {
    return(x+y);
}
```

Similarly, as in AS2.0, public methods of a class can be accessed using dot notation. For example, suppose a Car class has a public function move(). If car is an instance of Car, car.move() will invoke the move() function.

Properties

The splendid example of a class that we gave earlier is not very useful as it stands. Let us make it useful by making it do something, such as adding two numbers, for example.

This is how we might do it:

```javascript
package{
    public class DoSomething {
        private var x:Number = 2;
        private var y:Number = 3;
    }
}
```
public function DoSomething() {
    var z:Number;
    z = x + y;
    trace(z);  // write to output console
}

There is a lot happening here. First, we created properties x, y, and z. In AS3.0, properties need to be defined or declared first. This is done using the var keyword.

Note that we declare the properties x and y as private. This means that they are accessible within the class but not outside of it. “Within the class” means that any function in the class can access them.

Did you notice a difference in the way z is declared? There is no private keyword. That’s because it is declared within a function—in that case, it is accessible only within that function. So it cannot be declared as private because private properties are accessible from anywhere within the class.

How do you know whether to declare your properties and functions as private or public? The simple rule is to keep them private unless you really have to make them accessible from outside the class. This stops any outside code from potentially messing with your code.

Just as functions, public properties in AS3.0 can be accessed using dot notation. For example, ball.x gives the horizontal position of the ball object because x is a built-in public property of display objects such as MovieClips and Sprites (note that in AS2.0, this was denoted _x, with a leading underscore).

**Static methods and properties**

The methods and properties we’ve looked at pertain to objects, but it also is possible to have methods and properties for classes. This means that the class does not have to be instantiated to invoke the property or method.

An example is Math.random(), which is a class method from the Math class that generates a random number.

We use the static keyword to declare class level properties and methods. Static properties and methods are called using the class name, not an instance name. For example, suppose you have the following static method in a class called Physics:

```actionscript
static public function calcGravity(mass:Number, g:Number):Number {
    return(mass*g);
}
```

Physics.calcGravity(4, 9.8) would then give you the gravity force on a 4kg object on planet Earth.
Inheritance

An important concept in OOP is that of inheritance, which allows you to build new classes from existing classes. The new class is called a subclass, and the old class is called a superclass. This is done using the extends keyword.

For example, to create a subclass called Electron from a Particle class we’d do this:

```plaintext
package{
    public class Electron extends Particle {
        public function Electron() {
            // Electron code goes here.
        }
    }
}
```

The subclass Electron has access to all the non-private properties and methods of the Particle superclass, as well as those of its own. To invoke the superclass constructor, one simply calls super().

Note that the BouncingBall class extends the Sprite class. Every document class must extend either Sprite or MovieClip. To do so, one must first use an import statement to import the class or package, as done in BouncingBall.as.

ActionScript 3.0 language basics

Classes are clever concepts but, as the last section showed, they are pretty useless by themselves. In this section, we’ll review the code elements that do all the groundwork. Special emphasis is placed on their relevance to math and physics.

Variables and constants

Remember the var keyword that we used to declare properties? The keyword var stands for variable. A variable is a container that holds some value. Here value might mean different things. For example, the following variable x is defined to hold values that are numbers (we are omitting private keywords):

```plaintext
var x:Number;
```

Subsequently, x may only be assigned values that are numbers. For example:

```plaintext
x = 2;
```
This assignment can be done together with the following variable declaration or anywhere else in the code:

```actionscript
var x:Number = 2;
```

One can also perform arithmetic on `x`; for example, the following code multiplies `x` by a number, adds the result to another variable `y`, and assigns the result to a third variable `z`:

```actionscript
z = 2*x + y;
```

This resembles algebra, with some notable differences. The first difference is purely a matter of syntax: We use the operator `*` to multiply `2` and `x`. More about operators soon.

The second difference is more subtle and relates to the meaning of an assignment. Although the preceding code may look superficially like an algebraic equation, it is important to note that an assignment is not an equation. The difference can be highlighted by considering an assignment like this one:

```actionscript
x = x + 1;
```

If this were an algebraic equation, it would imply that `0 = 1`—an impossibility! Here, what it means is that we increase the value of `x` (whatever it is) by 1.

The word `variable` entails that we can change the value stored in a variable any time. If we want the value of a variable to be fixed, it might be a good idea to define it as a constant instead:

```actionscript
const gravity:Number = 1;
```

The value of a constant can be assigned only when it is defined. This is because declaring a `const` effectively prevents any subsequent assignment to be made in the rest of the code. There is no absolute need to use constants, but it may be a good idea to protect values that should not really be changed.

Variables in ActionScript can have values other than numeric values. The type of value that a variable can hold is called its `data type`.

### Data types

There are a number of different data types in AS3.0, and they may broadly be classified as `primitive` and `complex`. Primitive data types are the most basic types in AS3.0. ActionScript stores them in a manner that makes them especially memory- and speed-efficient. Complex data types, on the other hand, are made up of or reference primitives. They are more resource-intensive, but provide a lot of flexibility for programmers. Tables 2-1 and 2-2 list some of the more common data types in each category.

#### Table 2-1. Primitive Data Types

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>Has two possible values: <code>true</code> and <code>false</code>, or <code>1</code> and <code>0</code></td>
</tr>
<tr>
<td>int</td>
<td>32-bit integer</td>
</tr>
<tr>
<td>Number</td>
<td>64-bit double-precision floating-point number</td>
</tr>
</tbody>
</table>
### Table 2-1 continued.

### Table 2-2. Complex Data Types

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object</td>
<td>Defined by the <code>Object</code> class, which is the base class for all class definitions</td>
</tr>
<tr>
<td>Array</td>
<td>Holds a list of data of any type</td>
</tr>
<tr>
<td>Vector</td>
<td>Variant of Array that holds data of the same type</td>
</tr>
<tr>
<td>Sprite</td>
<td>A display object without a timeline</td>
</tr>
<tr>
<td>MovieClip</td>
<td>An animated movie clip display object</td>
</tr>
<tr>
<td>Bitmap</td>
<td>A non-animated bitmap display object</td>
</tr>
<tr>
<td>Shape</td>
<td>A nonanimated vector shape object</td>
</tr>
<tr>
<td>ByteArray</td>
<td>An array of binary data</td>
</tr>
<tr>
<td>TextField</td>
<td>A text field object</td>
</tr>
</tbody>
</table>

The Number, int, and uint primitive data types will be described in the subsection that follows, but let's say a few words about the remaining two primitive data types listed in Table 2-1: String and Boolean.

A String is group of characters. For example, the following

```actionscript
var str:String = “Hello there!”;
trace(str);
```

would give this output: `Hello there!`

Note that the value of a String must be enclosed within quotes (single or double).

A Boolean can have only one of two values: true or false. For example:

```actionscript
var bln:Boolean = false;
```

Note that the value true or false is not enclosed within quotes; it is not a string.

Among the complex data types, objects and arrays are particularly useful in terms of programming. The Array data type is described in a separate section that follows. The Object data type is defined by the `Object` class, which serves as the base class for all classes in ActionScript. The `Object` class by itself does not do much, but it is powerful because it is *dynamic*: it can be extended at runtime with new
properties and methods. Objects can also be used as associative arrays, which are arrays that store values using strings as keys (normal arrays store values using numeric indices —see the “Arrays” section that follows).

Because we’ll be using numeric data types and arrays an awful lot, let’s discuss them in a bit more detail.

**Numeric data types**

Numeric data types include `Number`, as discussed previously in the section “Variables and constants,” `int`, and `uint`.

The `Number` type is a double-precision 64-bit floating-point number according to the IEEE 754 specification. It is able to store both positive and negative real numbers (i.e. not only whole numbers, but those with fractional parts, too). The maximum value that `Number` can store is $1.8 \times 10^{308}$. Given that the number of atoms in the visible universe is estimated to be “only” $10^{80}$, this should be enough even for the biggest scientific calculations!

The `Number` class also includes the following special values: `NaN` (not a number), `POSITIVE_INFINITY`, and `NEGATIVE_INFINITY`.

`NaN` signifies that a numeric value has not been assigned. You’d get `NaN` if you look at the value of a `Number` variable that has not been given a value. You’d also get `NaN` as a result of a mathematical operation that produces non-real or undefined results (for example, by taking the square root of –1 or dividing 0 by 0).

Infinity is the result of dividing a non-zero number by 0. You will get positive or negative infinity depending on the sign of the number you are dividing by zero.

The `int` type is a signed 32-bit integer. This means it can represent both positive and negative integers (including zero) between –2,147,483,648 and 2,147,483,647.

The `uint` type is similar to `int` except that it can only be positive or 0. Its range of values is from 0 to 4,294,967,295. It is most often used for counting, as you might imagine.

**Arrays**

An array is an object that holds a collection of items. Suppose you have to keep track of a number of particles in your movie. You could do that by naming them individually as `particle1`, `particle2`, `particle3`, and so on. That might work fine if you have a few particles, but what if you have 100 or 10,000? That’s where an array comes in handy. You can just define an array called `particles`, for example, and put all the particles in there.

A simple way to create an array is by specifying the array elements as a comma-separated list enclosed by square brackets:

```javascript
var arr:Array = new Array();
arr = [2, 4, 6];
trace(arr[1]); // gives 4
```
As the preceding code snippet shows, the resulting array elements are then accessed by `arr[n]`, where `n` is an unsigned integer called the *array index*. Note that the array index starts from 0, so that the first array element is `arr[0]`.

There are several other ways of creating arrays. There are also lots of rules to do with the manipulation of arrays and array elements. We'll come across examples of those soon.

It is also possible to create multidimensional arrays. This is done by creating arrays whose elements are also arrays. The following example creates a two-dimensional array from two one-dimensional arrays:

```javascript
var xArr:Array = new Array();
var yArr:Array = new Array();
xArr = [1,2];
yArr = [3,4];
var zArr:Array = new Array(xArr,yArr);
trace(zArr[0][1]); // gives 2
trace(zArr[1][0]); // gives 3
```

Note that we've created the third array in a different way, by passing the array elements directly to the constructor of `Array()`.

It is possible to add different types of objects into the same array. That's because arrays in ActionScript are not typed, unlike in some other languages like C++ and Java. The `Vector` data type (available in Flash Player 10 or above) is a variant of `Array` that is typed. Vectors generally perform faster than arrays.

### Operators

You can perform basic arithmetic with numbers with the usual operators (`+`, `-`, `*` and `/`, respectively) for adding, subtracting, multiplying, and dividing numbers.

There are also a number of other, less obvious operators. The modulo operator `%` gives the remainder when a number is divided by another. The increment operator (`++`) increases the value of a number by 1, and the decrement operator (`--`) reduces the value of a number by 1.

```javascript
var x:int = 5;
var y:int = 3;
trace(x%y); // gives 2

var z:int;
z = x++; // assigns the value of x to z, then increments x
trace(z); // gives 5
z = ++x // increments the value of x, then assigns it to z
trace(z); // gives 7
```

Operators can also be combined with assignment. For example:

```javascript
var a:int = 1;
a = a + 1;
trace(a); // gives 2
```
a += 1; // shortened form of a = a + 1
trace(a); // gives 3
a = 4*a;
trace(a); // gives 12
a *= 4; // shortened form of a = a*4
trace(a); // gives 48

Math

Besides the basic operators described in the last section, the Math class contains many more mathematical functions.

Table 2-3 gives some common examples of Math functions and what they do. In the next chapter you will encounter many more Math methods, such as trigonometric, exponential, and logarithmic functions.

<table>
<thead>
<tr>
<th>Method</th>
<th>What it returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Math.abs(a:Number)</td>
<td>absolute value of a</td>
</tr>
<tr>
<td>Math.pow(a:Number,b:Number)</td>
<td>a to the power of b</td>
</tr>
<tr>
<td>Math.sqrt(a:Number)</td>
<td>square root of a</td>
</tr>
<tr>
<td>Math.ceil(a:Number)</td>
<td>smallest integer that is larger than a</td>
</tr>
<tr>
<td>Math.floor(a:Number)</td>
<td>largest integer that is smaller than a</td>
</tr>
<tr>
<td>Math.round(a:Number)</td>
<td>nearest integer to a</td>
</tr>
<tr>
<td>Math.max(a:Number,b:Number,c:Number,…)</td>
<td>largest of a, b, c, …</td>
</tr>
<tr>
<td>Math.min(a:Number,b:Number,c:Number,…)</td>
<td>smallest of a, b, c, …</td>
</tr>
<tr>
<td>Math.random()</td>
<td>a pseudo-random number n, where 0 &lt;= n &lt; 1</td>
</tr>
</tbody>
</table>

The last method, Math.random(), is an interesting one. It generates a random number between 0 and 1, including 0 but excluding 1. Strictly speaking, it is pseudo-random because it follows an algorithm. But it is good enough for most purposes you’re likely to use it for.

Here is an example of how to use the Math.random() method. In BouncingBallsRandom.as, we have made a simple modification so that each time the swf runs, the ball has a different initial velocity. We do this by the following two lines in init():

```javascript
vx = Math.random()*5;
vy = (Math.random()-0.5)*4;
```

The first line sets the initial horizontal speed to be between 0 and 5. The second line sets the vertical speed to be between –2 and 2. What does a negative vertical speed mean? It means a speed in the direction opposite to the direction of increasing y. Because in the Flash coordinate system y increases as
we go down (as we’ll see later in this chapter), negative vertical speed means that the object moves upward. So each time you start the swf you’ll see the ball initially move up or down with a different horizontal and vertical speed.

Logic

In any programming language, logic is an essential part of coding. Logic enables code to take different actions based on the outcome of some expression.

The simplest way to implement logic in ActionScript is through a basic if statement, which has the following structure:

```
if (logical expression){
    do this code
}
```

An if statement basically checks if a logical expression is true.

For example, in the BouncingBall.as code, there is the following logic:

```
if (ball.y > 350){
    vy *= -0.8;
}
```

This tests whether the ball’s vertical position is greater than 350 pixels and, if so, multiplies its vertical speed by –0.8. In this example, the logical expression to be tested is `ball.y > 350`, and `>` is a logical operator that means “greater than.”

Other commonly used logical operators include `<` (less than), `==` (equal to), `<=` (less than or equal to), `>=` (greater than or equal to), and `!=` (not equal to). There is also a strict equality operator `===`, which differs from the equality operator `==` in that it takes the data type into account when comparing two variables.

Care must be taken not to confuse the equality operator `==` with the assignment operator `=`. This is a common source of mistake and consequent debugging frustration!

There are also `&&` (AND) and `||` (OR) operators that enable you to combine conditions:

```
if (a < 10 || b < 20){
    c = a+b;
}
```

There are more elaborate forms of the if statement. The if else statement is of this form:

```
if (logical expression){
    do this if expression is true
} else {
    do this if expression is false
}
```
You can also use an if else if ... else statement to check for different possibilities:

```actionscript
if (a == 0) {
    do this if a is zero
} else if (a < 0) {
    do this if a is negative
} else if (a > 0) {
    do this if a is positive
} else {
    do this if a is NaN
}
```

Other logical constructs include the `switch` and the `ternary conditional operator`, but we won’t be using them much in this book.

Here is an exercise: modify the `BouncingBallRandom.as` code to recycle the ball so that when it disappears at the right boundary, it starts again at the initial location but with a new random velocity. The answer is in `BouncingBallRecycled.as`.

**Loops**

Just like logic, looping is an essential ingredient of programming. One of the things that make computers useful is their capability to repeat operations over and over again, much more quickly than humans, and without ever getting bored. They do it by **looping**.

In ActionScript there are several kinds of loops. We’ll review just a couple of those here.

The `for loop` is the one that we’ll make most use of. Here is an example of a for loop, used for summing the first 100 positive integers:

```actionscript
var sum:Number = 0;
for (var i:int = 1; i <= 100; i++) {
    sum += i;
}
trace(sum);
```

The first line initializes the value of the variable `sum` to 0. The next line sets up the loop—the variable `i` is a counter, set to start at 1 (you could start it from 0 or any other integer), up to and including 100, and told to increment by 1 (`i++`) on each step. So the loop executes 100 times, each time adding the current value of `i` to `sum`.

Looping over elements of an array is an especially useful technique. Suppose you want to animate five bouncing balls rather than one. To see how you’d do it, take a look at the code in `BouncingBalls.as`.

The main idea is to modify the `init()` function so that we create a bunch of balls, give each one a position and velocity (we’ve added `vx` and `vy` as public properties in the `Ball` class), and put them into an array named `balls` (which is defined as a private variable, so can be accessed later in another part of the code) using the `push()` method:

```actionscript
private function init():void {
    balls = new Array();
```
for (var i:uint=0; i<5; i++){
    var ball:Ball = new Ball();
    ball.x = 50;
    ball.y = 75;
    ball.vx = Math.random()*5;
    ball.vy = (Math.random()-0.5)*4;
    addChild(ball);
    balls.push(ball);
}
addEventLister(Event.ENTER_FRAME,onEachTimestep);
}

Naturally, the event handler is also modified to loop over all the balls:

private function onEachTimestep(evt:Event):void{
    for (var i:uint=0; i<5; i++){
        var ball:Ball = balls[i];
        ball.vy += g;
        ball.x += ball.vx;
        ball.y += ball.vy;
        if (ball.y > 350){
            ball.y = 350;
            ball.vy *= -0.8;
        }
    }
}

Don't worry that the balls just pass through each other when they meet. That's because your code doesn't know about collision detection yet! We'll fix that later.

Note that in order to use a for loop, you need to know exactly how many times you want to loop. If you don't, then there are other options, such as for ... in, for each ... in, and while loops. We won't describe the first two as we don't make use of them in this book.

In a while loop, you tell the loop to execute as long as some condition is true, no matter how many times you need to loop. The basic structure of a while loop is as follows:

while (some condition) {
    do something
}

For example, suppose you want to know the minimum number of consecutive integers you must sum, starting from 1, to obtain at least 1000. Here is a while loop to do this:

var sum:Number = 0;
var i:int = 1;
while (sum < 1000) {
    sum += i;
    i++;
}
trace(i-1);
You can also use a while loop in the same way as a for loop, to perform an operation a fixed number of times, for example, to sum the first 100 positive integers:

```actionscript
var sum:Number = 0;
var i:int = 1;
while (i <= 100) {
    sum += i;
    i++;
}
trace(sum);
```

Be careful with while loops —if the condition is always true, you’ll end up with an infinite loop and the code will never stop executing!

A variation is a `do ... while` loop, in which the condition is checked after the loop instead of before. This ensures that the code within the loop executes at least once:

```actionscript
do {
    do something
} while (some condition);
```

### Events in ActionScript 3.0

An *event* allows a given course of action to be replaced by a different course of action. Events contribute in a major way to making interactive media as interesting as they are. AS3.0 has quite an involved event framework. We won’t go into details of the event framework and how it works under the hood, but will just take a very practical approach by simply reviewing the code syntax for tracking and responding to events. Also we focus here only on a few examples that we’ll use frequently in our physics applications.

#### Event listeners and handlers

There are two aspects to event management: tracking events and responding to events. In AS3.0, *event listeners* “listen” to events, and *event handlers* take the appropriate action.

The syntax for setting up an event listener is as follows:

```actionscript
addEventListener(event_type, handler);
```

Here `handler` is simply a function that is called whenever an event of type `event_type` happens. `BouncingBall.as` contains an example:

```actionscript
addEventListener(Event.ENTER_FRAME, onEachTimestep);
```

Here the event type is `Event.ENTER_FRAME`, a static property of the `Event` class. It is automatically triggered at a rate set by the frame rate of the movie. So if the frame rate is 20 frames per second (fps), `Event.ENTER_FRAME` is triggered every 50 milliseconds. In response, the function `onEachTimestep()` is called and executed immediately each time the event is triggered. Note that the function `onEachTimestep` takes an argument of type `Event (evt:Event)`. 


You can also remove an event listener in exactly the same way, replacing `addEventListener` by `removeEventListener`. For example:

```javascript
removeEventListener(Event.ENTER_FRAME, onEachTimestep);
```

### Events and user interaction

There are many other classes that deal with events besides the `Event` class. For example, the `MouseEvent` class deals with mouse events and the `KeyboardEvent` class deals with, well, keyboard events.

Mouse and keyboard events are great because they allow the user to interact with the Flash movie. For example, suppose we want to pause the BouncingBall animation when the user clicks and holds down the mouse, and resume it when the mouse is released. This can be achieved easily using the `MouseEvent.MOVE_DOWN` and `MouseEvent.MOVE_UP` events. Just add the following lines to the `init()` method:

```javascript
addEventListener(MouseEvent.MOUSE_DOWN, stopAnim);
addEventListener(MouseEvent.MOUSE_UP, startAnim);
```

and include these event handlers:

```javascript
private function stopAnim(evt:MouseEvent):void{
    removeEventListener(Event.ENTER_FRAME, onEachTimestep);
}
```

```javascript
private function startAnim(evt:MouseEvent):void{
    addEventListener(Event.ENTER_FRAME, onEachTimestep);
}
```

The code is in `BouncingBallPause.as`. Note that you need to import the class `flash.events.MouseEvent`.

Note that the `addEventListener()` is actually a method of an object such as a sprite or movie clip. Therefore, you can do something like this:

```javascript
ball.addEventListener(MouseEvent.MOUSE_DOWN, stopAnim);
```

In this case, the animation will stop only if the ball object is pressed. If no object is specified, `stopAnim()` is called whenever the mouse is pressed anywhere on the stage.

### Drag and drop

Drag and drop is easy with AS3.0, using the aptly named `startDrag()` and `stopDrag()` methods. To illustrate the use of `startDrag()` and `stopDrag()`, we will modify the bouncing ball code yet again, so that now you can click the ball, move it anywhere on the stage, and then release it again.

To do this, make the following changes. First, remove the following line from `init()`:

```javascript
addEventListener(Event.ENTER_FRAME, onEachTimestep);
```
Replace it with the following two lines:

```actionscript
ball.addEventListener(MouseEvent.MOUSE_DOWN, drag);
ball.addEventListener(MouseEvent.MOUSE_UP, drop);
```

They set up listeners, and the corresponding event handlers are these:

```actionscript
private function drag(evt:MouseEvent):void{
    removeEventListener(Event.ENTER_FRAME,onEachTimestep);
    ball.startDrag();
}
private function drop(evt:MouseEvent):void{
    addEventListener(Event.ENTER_FRAME,onEachTimestep);
    ball.stopDrag();
}
```

You will also need to set the initial values of \(vx\) and \(vy\) to zero, the initial value of \(ball.y\) to 350 (so that it is initially stationary on the ground), and \(ball.x\) to any suitable value so that it is visible on the stage. The modified code is in `BouncingBallDrag.as`. Try it out!

**The Flash coordinate system**

In the real world things exist in space. In the Flash world the equivalent is that objects exist on the stage. To know how to position objects on the stage, it is necessary to understand the Flash coordinate system.

**2D coordinates**

Flash’s coordinate system is somewhat different from the usual Cartesian system of coordinates in math. In normal coordinate geometry, the \(x\)-coordinate runs from left to right, and the \(y\)-coordinate runs from bottom to top (see Figure 2-1b). In Flash, however, the \(y\)-coordinate runs in the opposite way, from top to bottom (see Figure 2-1a). The origin is in the top-left corner of the visible stage.

The usual Cartesian system is called a *right-handed coordinate system* because if you hold your right hand with your fingers closed and your thumb pointing out of the paper, your fingers will point from the positive \(x\)-axis to the positive \(y\)-axis. By implication, Flash’s coordinate system in 2D is a *left-handed coordinate system*. 

Another oddity in the Flash coordinate system is that angles are measured in a clockwise sense from the direction of the positive x-axis (see Figure 2-2a). The usual convention in math is that angles are measured counterclockwise from the positive x-axis (see Figure 2-2b).

3D in Flash

From Flash Player 10, 3D is supported natively. This means that it is possible to position and manipulate objects in 3D space using built-in AS3.0 properties and methods. To be sure, the capabilities of additional third-party 3D engines for Flash such as Papervision3D or Away3D far exceed the native 3D capabilities of Flash Player. However, the availability of native 3D means that it is now extremely easy to perform basic tasks in 3D.

3D means that there is a third axis: the z-axis. In Flash, the direction of increasing z points into the screen. Any display object such as a sprite has a property z that fixes the z-coordinate of the object on the screen. Note that the position on the screen matches the x- and y-coordinate values only if the z-coordinate is zero.

Figures 2-3a and 2-3b show the corresponding coordinate systems in 3D. Note that the usual right-handed Cartesian system used in math is frequently pictured with the y-axis pointing into the screen and the z-axis pointing upward. The 3D coordinate system used in Flash 10, however, has the y-axis pointing down (as in the 2D version) and the z-axis pointing into the screen. However, the two coordinate systems are, in a sense, equivalent because the Flash 10 3D coordinate system is also right-handed. In fact, if you rotate it by 90 degrees clockwise about the positive x-axis you recover the usual orientation of axes used in math.
Rotation is also possible around the three axes. An object’s rotation is specified by the properties `rotationX`, `rotationY`, and `rotationZ`, which are measured in degrees. The convention for the orientation of angles in 3D is again opposite to that in math, with angles about an axis being measured in a clockwise sense while looking in the positive direction along that axis.

To show how straightforward it is to incorporate basic 3D, we have modified the bouncing ball code to make the ball bounce off the four walls inside an invisible box. To achieve this, we invoked the third coordinate `ball.z` and introduced a corresponding velocity component `vz`. We initialize the values of these variables in `init()`. Then in the `onEachTimestep()` function, we update `ball.z` in exactly the same way as `ball.x`:

```
ball.z += vz;
```

The velocity component `vz` remains unchanged except when the ball bounces off a wall perpendicular to the z direction. This is because the z direction is also horizontal in this context, and gravity does not act horizontally.

The remaining additions in the code implement bouncing at the walls, following the same logic as that for bouncing at the ground (and taking into account the size of the ball).

This is what the updated `onEachTimestep()` method looks like:

```
private function onEachTimestep(evt:Event):void{

    vy += g; // gravity increases the vertical speed

    ball.x += vx; // horizontal speed increases horizontal position
    ball.y += vy; // vertical speed increases vertical position
    ball.z += vz;

    if (ball.y > 350){ // if ball hits the ground
        ball.y = 350;
        vy *= gfac; // its vertical velocity reverses/reduces
    }
    if (ball.x > 550 - w){
        ball.x = 550 - w;
        vx *= wfac;
    }
    if (ball.x < w){
```
Chapter 2

```javascript
ball.x = w;
vx *= wfac;
}
if (ball.z > 200){
    ball.z = 200;
vz *= wfac;
}
if (ball.z < -200){
    ball.z = -200;
vz *= wfac;
}
}
```

The full code is found in BouncingBall3D.as. There you are—a ball bouncing inside an invisible 3D box! Never mind if you can’t see the box yet; we’ll fix that soon.

**The Flash drawing API**

The Flash drawing application programming interface (API) allows you to draw things such as basic shapes and fills using ActionScript. Technically, this is possible because all display objects have a graphics property that can access the methods of the drawing API.

**Drawing lines and curves**

There are four methods that you need to know to be able to draw lines and curves: `lineStyle`, `moveTo`, `lineTo`, and `curveTo`.

- The `lineStyle(width, color, alpha)` method takes three arguments: the width of the line to be drawn, the line color, and its transparency.
- The `moveTo(x, y)` method moves the cursor to the specified location \((x, y)\) without drawing anything.
- The `lineTo(x, y)` method draws a straight line from the current location to the new location \((x, y)\) specified in its argument.
- The `curveTo(x1, y1, x2, y2)` method draws a curve from the current location ending at the new location specified by \((x2, y2)\) and with a control point at \((x1, y1)\). The control point determines the curvature of the curve.

For example, to draw a straight line from the point \((50, 100)\) to \((250, 400)\), you would do something like this:

```javascript
graphics.lineStyle(1,0x000000,1);
graphics.moveTo(50, 100);
graphics.lineTo(250, 400);
```

As an exercise, why not draw a grid using these methods? See the code in Grid.as.
There are also a few very useful methods for drawing primitive shapes such as circles and rectangles. For example, `drawCircle(x, y, radius)` draws a circle with a center at (x, y) and the specified radius; `drawRect(x, y, width, height)` draws a rectangle with the upper-left corner located at point (x, y) and with the specified width and height.

**Creating fills and gradients**

Producing fills is straightforward. The `beginFill(color, alpha)` method sets the fill to the specified color and alpha, to be applied to all objects drawn thereafter and until an `endFill()` is encountered. The following code snippet will produce a green rectangle without a border:

```actionscript
graphics.lineStyle(1,0x000000,0);
graphics.beginFill(0x00ff00);
graphics.drawRect(10, 10, 20, 30);
graphics.endFill();
```

Creating gradients is a bit trickier. To do the job, the method `beginGradientFill()` needs no fewer than eight arguments, although some have default values, so they don't necessarily need to be specified:

- **type**:String: a value that specifies which gradient type to use: `GradientType.LINEAR` or `GradientType.RADIAL`.
- **colors**:Array: An array of RGB hexadecimal color values used in the gradient.
- **alphas**:Array: An array of alpha values for each color in the `colors` array; valid values are from 0 to 1.
- **ratios**:Array: An array that specifies where to center each color; values range from 0 to 255.
- **matrix**:Matrix (default = null): A transformation matrix that tells Flash how to fill the shape.
- **spreadMethod**:String (default = "pad"): A value that specifies which spread method to use; chosen from `SpreadMethod.PAD`, `SpreadMethod.REFLECT`, or `SpreadMethod.REPEAT`.
- **interpolationMethod**:String (default = "rgb"): A value from the interpolation method that specifies which value to use: `InterpolationMethod.LINEAR_RGB` or `InterpolationMethod.RGB`.
- **focalPointRatio**:Number (default = 0): A number that controls the location of the focal point of the gradient; valid values are between -1 and 1.

The parameters you are most likely to need to change are the first five. Note that the last four parameters are optional because they have default values. A full description of all these parameters and what they do is outside the scope of this book, but we refer the reader to Adobe's online AS3.0 documentation for full details: [http://livedocs.adobe.com/flash/9.0/ActionScriptLangRefV3/](http://livedocs.adobe.com/flash/9.0/ActionScriptLangRefV3/).

Back in Chapter 1 we built a `Ball` class (that we didn't talk much about) that we used to create the ball object in `BouncingBall.as`. It's time now to take a look at the code in `Ball.as`. Based on the
In this section, most of the code should be self-evident. The less-obvious part of the code is the creation of the gradient fill, which is done with the following three lines:

```javascript
var matrix:Matrix = new Matrix();
matrix.createGradientBox(_radius,_radius,0,-_radius,-_radius/2);
graphics.beginGradientFill(GradientType.RADIAL,[0xffffff,_color],[1,1],[0,255],matrix);
```

After creating a matrix object, we used the `createGradientBox()` method, which takes five arguments: width, height, rotation, horizontal shift, and vertical shift. We set these parameters in relation to the radius of the ball to create a nice effect—feel free to experiment with alternative values. The final line invokes `graphics.beginGradientFill()` using a radial gradient, a two-color array (one of which is white), alphas of 1 for each, locating the center of each color at the two extremes 0 and 255, and the matrix object we created to fill the shape.

Example: Bouncing ball in a box

Okay, let's now put together what we discussed in the last two sections (Flash 3D and the drawing API) to create something useful: let's build that 3D box!

The code is in `BouncingBall3DWalls.as` and the final result is shown in Figure 2-4.

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

    public class BouncingBall3DWalls extends Sprite {
        private var g:Number=0.1; // acceleration due to gravity
        private var vx:Number; // initial horizontal speed
        private var vy:Number; // initial vertical speed
        private var vz:Number; // lateral speed
        private var ball:Ball;
        private var w:Number;
        private var gfac:Number=-0.99;
        private var wfac:Number=-0.99;

        public function BouncingBall3DWalls() {
            init();
        }

        private function init():void {
            setWall(0,200,250,300,400,-90); // left wall
            setWall(550,200,250,300,400,90); // right wall
            setWall(275,200,400,548,400,0); // far wall
            vx = Math.random()*5;
            vy = (Math.random()-0.5)*4;
            vz = 4;
            ball = new Ball();
            ball.x = 100;
            ball.y = 75;
            ball.z = 0;
        }
    }
}
```
Here’s how we do it. First we create a `Wall` class. If you look at the code in `Wall.as`, you see that this is very straightforward. We basically draw a filled rectangle using values for the width, height, color, and alpha that are supplied via the constructor. Then in the main code `BouncingBallBox`, we define a function
setWall() that basically creates a wall object, positions it, and rotates it in 3D space. This function setWall() takes eight arguments, the last two of which (color and alpha) are optional. The remaining six arguments set the location, size and orientation of the wall.

The final thing is to invoke setWall() four times in init() to create the four walls. Note that three of the walls are created before the ball instance is created and the fourth (the near side one) is created after. This is because there is no automatic depth sorting in Flash 3D, and objects are simply given a depth based on the order in which they are placed on the stage.

You need Flash Player 10 or above to view this movie and Flash CS4 or above to work with the source code.

Figure 2-4. A ball bouncing inside a 3D box

Producing animation using code

There are a number of ways to produce animation using code in AS3.0. For example, there is a Tween class that can be used to move, resize, and fade movie clips. However, our focus is to produce physics-based animation. We basically know how to move objects. What is needed is a method to measure the advancement of time. We need a clock.

Using the built-in frame stepping as a clock

We’ve already been using a clock: the ENTER_FRAME event. Let’s now take a closer look to see how good a time-keeper the ENTER_FRAME event is.

Suppose the frame rate of a movie is 50 fps. This means that, in theory, the Event.ENTER_FRAME event is triggered once every 1/50 seconds (every 20 milliseconds). We said “in theory” because the frame rate in the Flash Player is notoriously inaccurate and depends, for instance, on the capabilities of the machine running the movie. What effect can this inaccuracy have on the animation?
To answer this question, we'll need to do some simple math. Suppose that we want to move an object at a constant velocity of 100 pixels per second, and suppose that the frame rate of the movie is 50 fps. Let's increment the object's horizontal position by \( \text{vx} \) per frame as in the bouncing ball example:

\[
\text{private function onEachTimestep(evt:Event):void}\{
\text{ball.x += vx;}
\}
\]

In other words, \( \text{vx} \) is the horizontal velocity in units of pixels per frame. What value must we give \( \text{vx} \)? Well, the velocity in units of pixels per second is 100, and there are 50 frames per second. So the value of \( \text{vx} \) is 100/50 or 2. In general, we have the following relationship:

\[
(\text{Velocity in pixels per second}) = (\text{Velocity in pixels per frame}) \times (\text{Frame rate in fps})
\]

Okay, so if we set \( \text{vx} = 2 \), we should see the ball moving at 100 pixels per second. The speed in pixels per second is what we actually perceive on the screen. However, there is no guarantee that the frame rate at which the movie runs will be exactly the frame rate that is set. Suppose that your machine is slow or there are other things running on it, so that the actual frame rate is closer to 30 fps. This gives an actual velocity of only 60 pixels per second. Your object appears to be moving slower. What's more, if you republish your movie at a different frame rate, the velocity of the ball will be different.

Clearly we need a better alternative than a clock that depends on the actual frame rate at which the movie runs.

### Using the Timer class

So we just decided that EventENTER FRAME is not reliable for timekeeping. Is there an alternative? Yes, one alternative is the Timer class (found in the flash.utils package), together with the related TimeEvent class (found in the flash.events package).

What a Timer does is not actually that different from EventENTER FRAME. It “fires” an event at a specified rate. But Timer is a lot more reliable than EventENTER FRAME. To set up a timer you need to do the following. First, create a timer object:

\[
timer = \text{new Timer}(10, 1000);
\]

As you can see, a Timer() takes two arguments. The first one is the time delay, in milliseconds, between timer events. The second, optional, argument is the number of times that the timer event fires. If the second argument is set to 0, or not specified, the timer will fire indefinitely.

The next step is to set up an event listener that detects timer events:

\[
timer.addEventListener(TimerEvent.TIMER, onEachTimestep);
\]

and the corresponding event handler that responds to the timer event:

\[
\text{private function onEachTimestep(evt:TimerEvent):void}\{
\text{ball.x += vx;}
\}
\]
The other crucial thing is to start it! You see, unlike an enterframe, a timer does not exist or tick by default. So after you create it, you have to actually start it. This is easy enough with the `start()` method:

```javascript
timer.start();
```

Moreover, you can stop a timer at any time using the `stop()` method:

```javascript
timer.stop();
```

This is neat because it allows you to stop and restart your animation at will, without having to remove and add the event listener (which is not very elegant).

One thing you have to remember with a timer is that it does not update your screen. So, if your frame rate is lower than your timer rate, you may not see the update until after the next frame event. For example, suppose your timer delay is 10 milliseconds, but your frame rate is only 10 frames per second. Your frames will be updating every 0.1 seconds, or 100 milliseconds —every 10 timer events. This will make your animation appear “jumpy” despite the high timer rate. To fix this, use the `updateAfterEvent()` method to force a screen update after every timer event:

```javascript
private function onEachTimestep(evt:TimerEvent):void{
    ball.x += vx;
    evt.updateAfterEvent();
}
```

Although `Timer` is a big improvement on `Event.ENTER_FRAME`, it is not without its shortcomings. One of the main problems is that the actual time interval between timer events actually includes the time it takes to execute all the code within the event handler on top of the specified delay. If there is a lot of code in your event handler, it might mean your timer ticking rate is substantially slower than what you specified.

### Using `getTimer()` to compute elapsed time

Bad timekeeping can really mess up your physics. For really accurate timekeeping, what we need is a way to measure actual elapsed time. Keith Peters, in his book *Foundation ActionScript Animation*, gives a great way to do this: using the `getTimer()` function.

The `getTimer()` function, found in the `flash.utils` package, returns an integer equal to the number of milliseconds that have elapsed since the Flash Player has initialized. So, if you call `getTimer()` twice, in different parts of the code, and work out the difference in the returned value, that would give you the time that has elapsed between those two calls.

How does that help us with animation? The point is that we can calculate the actual time that has elapsed since the object’s position was last updated. Then we can use that time to calculate the amount by which to move it.
To see this in action, `TimerExample.as` animates the motion of a ball moving at constant horizontal velocity \(vx\). Here is the modified event handler:

```actionscript
private function onEachTimestep(evt:TimerEvent):void{
    // time elapsed in seconds since last call
    var dt:Number = (getTimer() - t)/1000;
    t = getTimer(); // reset t
    ball.x += vx*dt;
    evt.updateAfterEvent();
}
```

We added two lines here. The first line works out the time elapsed \(dt\) (this notation will become clear in the next chapter) since the last time the call to `onEachTimestep()` was made. Here \(t\) is a private variable initialized to `getTimer()` before the start of the animation. The next line resets \(t\) so that it can be used for the next call.

You’ll see also that we modified the code that updates the ball’s position. We’re now adding an amount \(vx*dt\) to the ball’s current position instead of \(vx\), as before. What’s going on here? Well, this is the whole point of calculating the elapsed time \(dt\). You see, previously we were interpreting the velocity \(vx\) as pixels moved per frame (if using `Event.ENTER_FRAME`) or per tick (if using `Timer`). The assumptions we were making in doing so were that the frames or ticks were of fixed duration, and that duration was just what we specified in the frame rate or timer delay parameter. As long as those assumptions work, we can use frames or timer ticks as a good proxy for time, and thinking of velocity in terms of pixels per frame or timer tick is a good idea. But here what we’re saying is this: let’s get back to thinking about velocity in the correct way, as pixels moved per second. Therefore, in \(dt\) seconds, the distance moved is \(vx*dt\), so that the new position is this:

```actionscript
ball.x += vx*dt;
```

The advantage of going back to the real meaning of velocity is that the motion is always computed correctly, independently of the frame rate or timer tick rate. This technique will come in handy when we start looking at more complex physics.

Here is the code for `TimerExample.as` in its entirety:

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

    public class TimerExample extends Sprite {
        private var vx:Number=100; // velocity in units of pixels per second
        private var ball:Ball;
        private var timer:Timer;
        private var t:int;

        public function TimerExample() {
            init();
        }
    }
```

```
private function init():void {
    createBall();
    setupTimer();
}

private function createBall():void{
    ball = new Ball();
    ball.x = 50;
    ball.y = 100;
    addChild(ball);
}

private function setupTimer():void{
    timer = new Timer(20);
    timer.addEventListener(TimerEvent.TIMER, onEachTimestep);
    timer.start();
    t = getTimer(); // initialise value of t
}

private function onEachTimestep(evt:TimerEvent):void{
    // time elapsed in seconds since last call
    var dt:Number = (getTimer() - t)/1000;
    t = getTimer(); // reset t
    ball.x += vx*dt;
    evt.updateAfterEvent();
}

Precalculating motion

As you’ll now be aware, the methods for animating objects with code we’re using work by calculating updates to the object’s position “on the fly.” It might also be possible to precalculate the motion of an object and animate it afterward. This can be done by using a for or while loop to represent time-stepping, calculating the particles position at each step and saving the position coordinates in an array. You can even save the values in an external file (if using Adobe AIR) to be used by another movie.

Why would you want to do that? It can be useful, for example, if calculations take too long to perform and cannot fit within a reasonable frame rate.

The downside of this method is that it doesn’t work with interactivity. Because user interaction is usually an important aspect of physics-based applications, we won’t generally use this approach.

Collision detection

Collisions are an inevitable fact of life in an animated world. So far, the movies we created do not know how to detect when two objects have collided or what to do when they do. This section focuses on the first issue: collision detection. Responding properly to collisions will be treated in detail in Chapter 11.
In general, collision detection can be a tricky problem, but if the objects to be tested are rectangular in shape, there is a very simple built-in method: hitTestObject().

**Using the hitTestObject() method**

The hitTestObject() method determines whether two display objects (such as sprites or movie clips) are in collision:

```actionscript
object1.hitTestObject(object2);
```

The outcome is a Boolean: it returns true if they do and false if they don’t. So you’d typically use it as a condition in an if statement, writing code to handle the collision if it returns true.

What hitTestObject() actually does is to check whether the bounding boxes of the two display objects overlap or not. If the objects are rectangular in shape, obviously their bounding boxes coincide with their visible area, and hitTestObject() works like a dream. However, if one or both of the objects is any other shape (for example, a circle), that wouldn’t be true, and hitTestObject() might return true, even in some cases when the objects are not actually touching.

**Using the hitTestPoint() method**

The hitTestPoint() method checks whether a given point is touching a display object. It takes two mandatory parameters and one optional parameter and returns a Boolean:

```actionscript
object.hitTestPoint(x:Number, y:Number, shapeFlag:Boolean = false);
```

The first two parameters specify the coordinates of the point to be tested against the object. The shapeFlag option tells Flash whether to test against the visible area of the object (if set to true) or against its bounding box (if false, the default).

With shapeFlag=true, this method is especially handy if one of the objects is quite small (think projectile), even if the other object has a complicated shape (think spaceship): boom!

**Distance-based collision detection**

This method involves checking the distance between two objects explicitly and is simplest to formulate for circular objects. We have already used a variant of this method when checking for collisions of a bouncing ball with the ground or walls. Basically, you work out, using simple geometry, the minimum distance the objects can be before they touch. Let’s assume we have a circular ball with the registration point located at its center. Then the minimum distance the ball can be from a wall is equal to its radius. For example, to check if a ball hits a wall on the right side, the test would look like the following, where _wallx is the location of the wall and _radius is the radius of the ball:

```actionscript
if (ball.x >= _wallx - _radius){
    code to handle collision
}
```
If you’re testing whether two circles (of radii $r_1$ and $r_2$) are colliding, you need to check if the distance between their centers is less than or equal to $r_1 + r_2$. Why the sum of their radii? Because that’s always how far their centers are whenever they’re touching. Okay, but how do we calculate the distance between the centers of the circles? This boils down to a common problem in elementary geometry—the formula for the distance between two points. Lucky for us, a Greek guy called Pythagoras worked that out a long time ago. It’s called Pythagoras’s Theorem. We’ll take a closer look at that theorem in the next chapter. For now, here is the formula:

$$\text{dist} = \text{Math.sqrt}((x_2 - x_1) \times (x_2 - x_1) + (y_2 - y_1) \times (y_2 - y_1));$$

This gives the distance between two points with coordinates $(x_1, y_1)$ and $(x_2, y_2)$. So you take the difference between the x-coordinates and square it, do the same with the y-coordinates, add the results, and then take the square root.

Now if you’re talking about two circles, $(x_1, y_1)$ and $(x_2, y_2)$ will be the coordinates of their centers, and are in fact equal to their x and y properties if their registration points are in the center. So, here is our condition for detecting collisions between two circles:

$$\text{if (dist} \leq (r_1 + r_2)) {$$
$$\text{code to handle collision}$$
$$}\$$

Before you rush off to code this up, you can simplify things a bit. That $\text{Math.sqrt}()$ is not entirely necessary, and could eat up lots of CPU time in a timer or enterframe loop. And things would get much worse if multiple objects were involved. The trick is to check for the square of $\text{dist}$ instead:

$$\text{rSquare} = (r_1 + r_2) \times (r_1 + r_2);$$

$$\text{distSquare} = (x_2 - x_1) \times (x_2 - x_1) + (y_2 - y_1) \times (y_2 - y_1);$$
$$\text{if (distSquare} \leq \text{rSquare}) {$$
$$\text{code to handle collision}$$
$$}\$$

Note that if you only have two objects, or if all the objects have the same radius, the square $(r_1 + r_2)^2$ needs to be evaluated only once, outside of the time-stepping loop.

The formula for $\text{distSquare}$ can easily be generalized to 3D:

$$\text{distSquare} = (x_2 - x_1) \times (x_2 - x_1) + (y_2 - y_1) \times (y_2 - y_1) + (z_2 - z_1) \times (z_2 - z_1);$$

The formula for $\text{rSquare}$ remains the same in 3D, of course.

When multiple objects are involved, you still need to check every pair of objects using the same formula.

The file $\text{CollidingBalls.as}$ modifies $\text{BouncingBalls.as}$ to implement this collision-detection method. Take a look at the code and have a play with the swf, but don’t worry too much about the physics to deal with the collisions for now. We shall handle that in Chapter 11.
Advanced collision detection

You are now well equipped to detect collisions between rectangular objects (using HitTestObject), a point or small object and another object of arbitrary shape (using HitTestPoint with the shapeFlag option set to true), and two circular or spherical objects (using Pythagoras’s Theorem). But what about more complex shapes?

Several clever techniques exist for testing for collisions between irregular objects. One of them is by looking at individual pixels of BitmapData objects. We refer the interested reader to Chapter 1 of Keith Peters’s *Advanced ActionScript 3.0 Animation*.

Summary

Wow! This chapter has been a whirlwind tour through vast stretches of Flashland. Hopefully, you’ve now gained an appreciation of how different aspects of ActionScript can be useful for physics-based animation.

If you’ve struggled with any of the material in this chapter, we highly recommend that you brush up on your knowledge of Flash and ActionScript 3.0. Here are a couple of books that we particularly recommend for beginners: