Copyright 2005, Scott Leutenegger

Chapter 3: Bouncing Gargoyles: Branching Control Structures

So now we can make a ball move across the screen, and in last chapter's exercises you learned how to move the ball up or down as well as left or right. But the animation is pretty boring if the ball moves off the screen and never comes back. We need some way to make it bounce back. We want some way to tell the program: "If the ball moves to the end of the screen, have the ball reverse direction". Almost all computer programming languages provide a way to do code this type of control with something called "control structures". Control structures allow control over program execution flow. Examples in ActionScript include if/else statements, switch statements, for loops, and while loops. We will see all of these but we start with the one that allow execution to branch into different directions depending on evaluated conditions.

3.1: MovieClip Class and Objects

Before we go on to learning about if/else, we need to learn a bit of Flash/ActionScript specific information. The MovieClip class is heavily used in ActionScript game programming. The MovieClip class is a built in ActionScript class. This means that the environment has made this class available for us to use. A *class* is a description of an entity that contains data members and methods. Methods are ways to interact with the class. On *object* is a specific instance of a class.

Lets consider an example. Lets say I define an *Automobile* class. The class needs to define important data that belong to all cars such as current value, length, height, mpg, weight, number of doors, number of cylinders, color, and so on. The class would also have associated with it *methods* to access the data and/or carry out some action relevant to the object. One method for our automobile class might be *calculateTaxRate* which calculates the annual tax rate based on some function of current value, weight, and mgp. Another possible method might be *getWeight* which simply returns the weight of a car.

The class is like a "blue print", "template", or "cookie cutter". Just as a cookie cutter specifies the shape of the actual cookies produced, a class specifies the contents of the actual objects produced with it. More specifically, the class specifies the data and methods that an object will have. Many objects can be created from the class. Each created object is said to be an instance of the class. For example I could have millions of automobile objects (say I am the DMV and keeping track of autos across the nation) yet all are modeled on the *Automobile* class.

Later in the book we will construct our own classes and our own objects from those classes. For now, we just need to understand that ActionScript has built in classes and we can use those classes to create objects.

The MovieClip class is the one we will focus on. If you go to the Help pages of flash, search for "MovieClip" and then click on the MovieClip class page. Lots of info here! Look at the property summary. You will see that MovieClip objects have many properties, also called data members, including:

_x	x-coordinate of the MC
_y	y-coordinate of the MC
_width	Width of MC in pixels
_height	Height of MC in pixels
_rotation	Degree of MC image rotation
_xscale	x-dimension scaling factor in percent (i.e. 50 = 50%)
_yscale	y-dimension scaling factor in percent
_alpha	The MC's transparency in percent (i.e. 10 = 10%)

We have been using some of these data members already without knowing what they were, again, just sort of intuitive. Lets look at some code and explain in detail. First, create a MovieClip in a library. To make it a tad more interesting then a ball download the simple gargoyle drawing:

www.cs.du.edu/~leut/1671/FlashFiles/gargoyle.jpg

Save the above drawing in two a file and name it "gargoyle.jpg". Now import this drawing, convert it into a MovieClip, and experiment with a few MovieClip data members. Do the following:

1. Open Flash and create a new project
2. Import the gargoyle into flash: File->Import->Import To Stage. Use the file browser that pops up to select the file.
3. Save it to the library: Modify->Convert To Symbol. Change the name to "gargoyle" and check the "Export for ActionScript" box below Linkage, then hit okay.
4. Delete the drawing from the stage.

Now open the library (by hitting ctrl-L or Window->Library. You will see two entries named "gargoyle". Look at the "kind" field. One is a MovieClip (that you just created) and the other is a bitmap (that was put in the library when you imported the file). Notice also that the "Linkage" column has an entry of "Export:gargoyle" for the gargoyle MovieClip.

Lets write some code to play with the gargoyle. If you have not done so yet, close the library. Your stage should be empty now. Add the following ActionScript code to the first frame:

```
var g1:MovieClip ;
g1 = attachMovie("gargoyle","g1",1) ;

g1._x = 100 ;
g1._y = 100 ;
```

Now we can explain what is happening. The first line of code creates a variable named g1 that can hold an object of type MovieClip. The second line creates a MovieClip instance and assigns it to variable g1. The third and fourth lines set the _x and _y data member values of the MovieClip object held in g1 to 100 and 100 respectively.

If you had any problems creating this, you can also download the file:

www.cs.du.edu/~leut/1671/FlashFiles/c3_gargoyle1.fla

Note, the first and second line can be combined into a single line that creates the variable and assigns an instance of the MovieClip to it:

```
var g1:MovieClip = attachMovie("gargoyle","g1",1) ;
```

When you run this code you will notice that the drawing goes outside of the stage. That is because it is too big for the location at which is was place. You can do two things to fix this: reduce the size of the image so it fits, or move the image down and to the right. To reduce the size you can change the _xscale and _yscale data member values:

```
g1._xscale = 50 ;
g1._yscale = 50 ;
```

The above two lines of code will scale the image down to 50% of its original size in both the x and y-dimension. In addition to changing the scale of the MovieClip image, you can also change the rotation and the transparency:

```
g1._rotation = 50 ;
g1._alpha = 50 ;
```

The first line above will rotate the image 50 degrees clockwise. The second line will reduce the opacity of the image to 50% of the original.

The following code, found in www.cs.du.edu/~leut/1671/FlashFiles/c3_gargoyle2.fla , creates six instances of the gargoyle. The three images on top row are reduced to 25% of the original size with the second and third in the row rotated. The second row images are reduced to 50% of original size, and then going from left to right the opacity is reduced:

```
var g1:MovieClip = attachMovie("gargoyle","g1",1) ;
g1._x = 100 ;
g1._y = 100 ;
g1._xscale = 25 ;
g1._yscale = 25 ;

var g2:MovieClip = attachMovie("gargoyle","g2",2) ;
g2._x = 200 ;
g2._y = 100 ;
g2._xscale = 25 ;
g2._yscale = 25 ;
g2._rotation = 30 ;

var g3:MovieClip = attachMovie("gargoyle","g3",3) ;
g3._x = 300 ;
g3._y = 100 ;
g3._xscale = 25 ;
g3._yscale = 25 ;
g3._rotation = 60 ;

var g4:MovieClip = attachMovie("gargoyle","g4",4) ;
g4._x = 100 ;
g4._y = 300 ;
g4._xscale = 50 ;
g4._yscale = 50 ;

var g5:MovieClip = attachMovie("gargoyle","g5",5) ;
g5._x = 300 ;
g5._y = 300 ;
g5._xscale = 50 ;
g5._yscale = 50 ;
g5._alpha = 30 ;

var g6:MovieClip = attachMovie("gargoyle","g6",6) ;
g6._x = 500 ;
g6._y = 300 ;
g6._xscale = 50 ;
g6._yscale = 50 ;
g6._alpha = 10 ;
```

A key concept here is that there are 6 objects of type MovieClip. MovieClip is the general template/cookie cutter, and all 6 objects have the same data members and methods, but, they are unique objects and each can have different data member values.

Note that the second and third values inside the parenthesis of the attachMovie method all differ. These values in the parenthesis are called “arguments”, as we will learn in

Chapter ZZZ. The second argument needs to be a unique name for ActionScript to internally keep track of all the created objects. The third argument needs to be unique also, this is the Z-order or depth order that the object is placed in. Objects with a higher depth order are "in front" of objects with a lower order[1]

The MovieClip class has many other members and methods that you can explore later. Again, to get more information about the class use Help inside of Flash and search for MovieClip.

3.1.1: Vector Art versus Bitmaps

In the above example you imported a jpg file and then used it. In general this is a bad idea, it is better to use vector graphics. Vector graphics are lines and curves. The file actually stores the information describing the lines and curves. As a result the can be scaled without looking "pixilated". Bitmap graphics on the other hand are a grid of pixels. Bitmaps do not look good when they are scaled too large. Another disadvantage of bitmaps is size. Usually they are much larger than vector graphics, even in compressed form such as jpeg and gif.

When you draw pictures in flash they are stored as vector graphics, this is the method of choice for flash. If you have an image you want to use, say a photo or a scanned drawing, you can import into flash, put a layer on top of the photo, and then trace the parts of the photo you want. Then delete the layer with the photo and remember to remove the photo from the library. You can also import the file, then use the trace function found in Modify->Bitmap->Trace.

3.2: if / else

The if/else statement allows the execution of the code to be dependent on the current state of the program. The general form is:

```
if  (condition)
        { statement1 }
else
        { statement2 }
```

[1] If you try to use the same depth value as an earlier object, Flash will remove the earlier object and replace it with the new one.

What this means if the condition specified above by "(condition)" is true, then execute the code in statement1. If the condition is not true, then execute the code in statement2. Note, statement1 and statement2 can be a single statement or a statement block. A statement block is two or more statements grouped together by curly braces.

3.2a: If

The if / else statement is not required to have an else part, in which case it looks like:

```
if  (condition)
        { statement1 }
```

Lets start with the simpler case of the if without the else clause. Consider the following code:

```
var num1:Number = 5 ;

if (num1 < 10)
        trace("Yes, the number is less than 10") ;
trace("Goodbye") ;
```

In the above example, the condition to be evaluated is " (num1 < 10) ". If the content of variable num1 is less than 10 then the following statement is executed. If not, then the following statement is skipped. So, the output from above would be:

```
Yes, the number is less than 10
Goodbye
```

Lets change the variable creation statement to initialize the content to be greater than 10:

```
var num1:Number = 20 ;

if (num1 < 10)
        trace("Yes, the number is less than 10") ;
trace("Goodbye") ;
```

In this case the condition is false, so the first trace statement is skipped and only "Goodbye" is printed out. Notice that we have indented the first trace statement. The code works identically with or without this indentation, but it makes the code easier for a human to read if the indentation is included. This is considered good "programming style".

As mentioned above, we can enclose a statement inside curly braces, and if we put multiple statements inside the curly braces we can have all the statements executed as a block. First consider:

```
var num1:Number = 5 ;

if (num1 < 10) {
        trace("Yes, the number is less than 10") ;
}
trace("Goodbye") ;
```

The only changes are that we have changed the variable assignment value back to 5 and we have added curly braces around the first trace statement. The braces have no effect on the program flow. If the condition is true, everything inside the curly braces is executed. Now consider what happens if we add another statement inside the curly braces:

```
var num1:Number = 5;

if (num1 < 10) {
        trace("Yes, the number is less than 10") ;
        trace("Good deal, cuz less than 10 is cool") ;
}
trace("Goodbye") ;
```

In this case all three trace statements are printed. If the value is changed to 15 to start only the last trace statement is printer. The two statements inside the curly braces are treated as a single unit: either both are executed or neither. This extends to as many statements as you want to stick inside the curly braces.

Okay, that was a fun digression, but what does this have to do with our goal of bouncing balls? Simple: we need an if statement to change the direction of the ball. Consider the following code (www.cs.du.edu/~leut/1671/FlashFiles/c3_leftRight1.fla) :

```
var theBall:MovieClip = attachMovie("ball","b1",1) ;
theBall._x = 50 ;
theBall._y = 200 ;
```

```
var xIncrement:Number = 6 ;

onEnterFrame = function() {

        theBall._x += xIncrement ;

        if (theBall._x > 400)
          xIncrement *= -1 ;

}
```

You have seen the first three lines above in the last chapter. They create a variable of type MovieClip and attach a MovieClip object called ball (which needs to be in the library) and then position it at pixel location (50,200). The next line of code creates a variable of type Number to hold the x-coordinate increment value. This variable is used inside the onEnterFrame function.

Remember that the onEnterFrame function is executed every time a frame is entered.[2] A function is a block of code, just like a statement block above. There are more differences that will be explained in chapter ZZZ, but for now realize that all the code between the curly braces is treated as one unit. The code in the function has two parts. First, the current x-coordinate of the ball, denoted as *theBall._x* ,is incremented. The amount that *theBall._x* is incremented is the current value in variable *xIncrement.* If the value is currently positive, then the ball is moved to the right. If the value is negative then the ball moves to the left. So, on every frame entry the ball's location is moved by an amount equal to the value in variable *xIncrement*. Initially the value is positive, and hence the ball keeps being moved to the right. The next snippet of code is what changes the direction of the ball:

```
if (theBall._x > 400)
        xIncrement *= -1 ;
```

Each time the function is executed the condition "theBall._x > 400" is checked. If true, then the next line of code is executed. When executed the variable value is negated so that on the next entry the to the function, and then every subsequent function entry, the location of the ball is moved to the left. Eventually the ball's location becomes less than zero and the ball moves off the stage never to return.

[2] Actually, to be technically correct, the onEnterFrame function is not actually executed when a frame is entered, but rather it is executed at the same rate as the rate that frames are entered. The effect is the same,

This is a great first step, but the ball is not moving back and forth, it needs to bounce back when it hits the left side also. How do we do this? With yet another if statement of course! Consider the following modification to the onEnterFrame function

(www.cs.du.edu/~leut/1671/FlashFiles/c3_leftRight2.fla):

```
onEnterFrame = function() {

        theBall._x += xIncrement ;

        if (theBall._x > 400)
          xIncrement *= -1 ;

        if (theBall._x < 0)
          xIncrement *= -1 ;
}
```

Now when the ball gets to an x-location less than zero the value of the increment is negated again, making it positive, so that on subsequent executions of the onEnterFrame function the ball is moved to the right, that is until it gets to 400 again and then the direction is reversed.

Finally, notice how the ball does not make it to the right-most part of the stage. That is because the stage is wider than 400 pixels. You can get the correct number from the properties panel, or, you can use the built in width member of Stage object

(www.cs.du.edu/~leut/1671/FlashFiles/c3_leftRight3.fla):

```
if (theBall._x > Stage.width)
          xIncrement *= -1 ;
```

The Stage object contains information about the stage such as Stage.width and Stage.height. Now the ball bounces back and forth the entire width of the stage. If you go into the properties panel and change the size of the stage, the ball will bounce at the right place because you specified Stage.width instead a specific number.

You may want to make the ball bounce up and down instead of left-right. This can be done by changing the ball's _y value. Notice that we use Stage.height in this example.

(www.cs.du.edu/~leut/1671/FlashFiles/c3_upDown3.fla):

```
var yIncrement:Number = 6 ;

onEnterFrame = function() {

        theBall._y += yIncrement ;

        if (theBall._y > Stage.height)
          yIncrement *= -1 ;

        if (theBall._y < 0)
          yIncrement *= -1 ;
}
```

3.2b: If With Else

In the preceding section we saw how to make code execution dependent on some condition being true. Often we want to execute some code if the condition is true, and other code if the condition is false. Consider the following code:

```
var num1:Number = 15 ;

if (num1 < 10)
   trace("num1 is less than 10") ;
else
    trace("num1 is greater than or equal to 10") ;
trace("bye") ;
```

In this case two lines are printed out: "num1 is greater than or equal to 10" and "bye".

As we have seen we can use braces to create statement blocks like:

```
var num1:Number = 15 ;

if (num1 < 10) {
        trace("num1 is less than 10") ;
        trace("definitely less than 10") ;
}
else {
        trace("num1 is greater than or equal to 10") ;
        trace("definitely greater than or equal to 10") ;
}
trace("bye") ;
```

In this case the 3rd, 4th, and 5th trace statements are executed. If variable num1 is initialized with a value less than 10 the 1st, 2nd, and 5th trace statements will be executed.

Consider a ball example with compound statements

(www.cs.du.edu/~leut/1671/FlashFiles/c3_leftRight4.fla) :

```
var xIncrement:Number = 6 ;

onEnterFrame = function() {

        theBall._x += xIncrement ;

        if (theBall._x > Stage.width) {
          xIncrement *= -1 ;
          xIncrement *= 3 ;
        }

        if (theBall._x < 0) {
          xIncrement *= -1 ;
          xIncrement /= 3 ;
        }
}
```

Here two things happen when the ball's location exceeds the stage width: the xIncrement variable is negated and the xIncrement variable is increase three times. Two things happen when the ball's location is less than 0: the xIncrement variable is negated and the xIncrement variable's value is dived by three. The effect is that the ball moves to the left three times faster than it does to the right.

Note: this example is actually contrived, because you could get the exact same effect without a statement block by simple replacing as follows:

```
onEnterFrame = function() {

        theBall._x += xIncrement ;

        if (theBall._x > Stage.width)
          xIncrement *= -3 ;

        if (theBall._x < 0)
          xIncrement /= -3 ;

}
```

Sometimes though you really need a statement block. Consider the following example

(www.cs.du.edu/~leut/1671/FlashFiles/c3_leftRight5.fla):

```
var xIncrement:Number = 6 ;

onEnterFrame = function() {

        theBall._x += xIncrement ;

        if (theBall._x > Stage.width) {
          xIncrement *= -3 ;
          theBall._xscale *= 2 ;
          theBall._yscale *= 2 ;

        }

        if (theBall._x < 0) {
          xIncrement /= -3 ;
          theBall._xscale /= 2 ;
          theBall._yscale /= 2 ;
        }
}
```

In this example the ball moves to the left 3 times faster than to the right as before, but now the ball is also twice as big as it moves to the left. This is achieved by using the _xscale and _yscale members of the ball MovieClip object. Statement blocks are necessary to change both of these properties at once.

Notice how we used the _xscale and _yscale properties of the MovieClip object. When we multiple or divide these values it scales the x or y dimensions of the MovieClip by the amount that we specified.

3.3: Boolean conditions, comparison operators, logical operators

In the previous two sections we used something called "Boolean conditions" and you probably did not even notice. You probably did not notice because they are fairly intuitive. A Boolean condition is just a fancy name for a condition that evaluates to TRUE or FALSE. The condition is evaluated using some sort of "comparison operator". Examples from above included:

- (theBall._x < 0)
- (num1 < 10)
- (theBall._x > Stage.width)

Each on of these conditions evaluates to true or false. In each of these the comparison operator was either “<” or “>”. Comparison operators include:

- <   (less than)
- >   (greater than)
- <=   (less than or equal)
- >=   (greater than or equal)
- <>   (not equal)
- !=   (not equal)
- ==   (equal)

Comparison operators are used in Boolean condition tests. Some examples include:

1. (num1 < 10)
2. (5 >= numDogs)
3. (2 == numChildren)
4. (5 <= 2)
5. (num2 != 5)

Example 1 evaluates to TRUE if the contents of variable num1 are less than 10, otherwise the condition evaluates as FALSE. Example 2 evaluates to TRUE if the value of variable numDogs is less than or equal to 5. Example 3 evaluates to TRUE if the value of variable numChildren equals 2. Example 4 always evaluates to FALSE. Example 5 evaluates to TRUE if the value of variable num2 does not equal 2.

WARNING: note that the equal comparison operator is two equal signs, not one. One equal sign is the assignment operator.

Consider the following code:

```
var num1:Number = 5 ;
var num2:Number = 10 ;

if (num1 == num2)
   trace("num1 equals num2")

if (num1 = num2)
```

```
trace("num1 = num2") ;
```

This code prints out “num1 = num2” even though the contents of variable num1 do NOT equal the contents of variable num2. The reason is the second if condition contains the assignment operator =, not the equality comparison operator ==. The assignment operator assigns the content of the right hand operand into the left hand operand and returns “true” if the assignment succeed. Hence, if you use a single equal sign when you meant a double equal sign, the condition will always evaluate as true.

Boolean conditions can also be made up of compound tests tied together by logical operators. The three logical operators are **&&** and **||**. The first is AND, the second is OR. The **&&** operator evaluates to true if both operands are true, otherwise the operator returns false. The **||** operator returns true if one or both of the operands are true. If both operands are false then **||** operator returns false. Consider the following examples:

1. ((num1 < 5) && (num1 > 2))
2. ((num1 > 50) || (num1 < 20))
3. ((theBall._x < 0) && (theBall._y >= 50))

The first example evaluates to TRUE if num1 is greater than 2 AND num1 is less than 5. The second evaluates to TRUE if num1 is either greater than 50, or, num1 is less than 20. The third evaluates to TRUE if the _x value of theBall is less than zero AND the _y value of theBall is greater than or equal to 50.

Logical expressions can be put together using comparison expressions and logical operators as building blocks. Assuming the obvious variable declarations precede these expressions, each of the following examples are legal expressions:

1. ((num1 < num2) && (num2 < num3) && (num3 < num4))
2. (((num1 < num2) && (num2 < num3)) || ((num3 < num2) && (num2 < num1)))
3. ((theBall._x < 50) && (theBall._x > 0)) || (theBall._y > 50)

The first expression evaluates to true if the contents of variable num1 are less than the contents of num2 which in turn are less than the contents of num3 which in turn are less then the contents of num4.

The second example evaluates to true if either the contents of num1 are less than the contents of num2 which in turn are less than the contents of num3, or, the converse is

true, i.e. num3 < num2 < num1. Note, you can NOT write a three way comparison operator like:

```
if (num1 < num2 < num3)
        trace("This is bogus!") ;
```

ActionScript only allows comparison expressions to have one operator and two operands. If you want to achieve the equivalent of above, you need to write two comparisons and tie them together with the logical **&&** operator, as done in the second example above. Notice the use of parenthesis. In the second example the format (A) || (B) is used, where (A) is actually ((num1 < num2) && (num2 < num3)) and (B) is actually ((num3 < num2) && (num2 < num1)).

The third example above evaluates to true if either the ball is located at a y-coordinate value greater than 50, or if the ball's x-coordinate value is between 0 and 50.

3.4: Nested If/else

Remember the general if /else statement format:

```
if  (condition)
        { statement block 1}
else
        {statement block 2}
```

Remember that we said statement blocks could be one statement or many. Not only can there be many statements in a block, but they can contain another if/else statement. Consider the following chunk of code:

```
var num1:Number = 3;
var num2:Number = 5 ;
var num3:Number = 2 ;

if (num1 <= num2) {
        if (num2 <= num3)
                trace("num1 <= num2 <= num3") ;
        else
                if (num1 <= num3)
                        trace("num1 <= num3 <= num2") ;
                else
                        trace("num3 <= num1 <= num2") ;
```

```
}
else {
        if (num1 <= num3)
                trace("num2 <= num1 <= num3") ;
        else
                if (num2 <= num3)
                        trace("num2 <= num3 <= num1") ;
                else
                        trace("num3 <= num2 <= num1" ) ;
}
```

Given there are three variables being compared, there are 6 possible orderings:

1. num1 <= num2 <= num3
2. num1 <= num3 <= num2
3. num2 <= num1 < = num3
4. num2 <= num3 <= num1
5. num3 <= num1 < = num2
6. num3 <= num2 <= num1

Convince yourself that the above code correctly determines which of the 6 cases is true and prints it out. Note, by using logical operators, there is a simpler way to do the same test:

```
if ( (num1 <= num2) && (num2 <= num3) )
        trace("num1 <= num2 <= num3" ) ;

if ( (num1 <= num3) && (num3 <= num2) )
        trace("num1 <= num3 <= num2" ) ;

if ( (num2 <= num1) && (num1 <= num3) )
        trace("num2 <= num1 <= num3" ) ;

if ( (num2 <= num3) && (num3 <= num1) )
        trace("num2 <= num3 <= num1" ) ;

if ( (num3 <= num1) && (num1 <= num2) )
        trace("num3 <= num1 <= num2" ) ;

if ( (num3 <= num2) && (num2 <= num1) )
        trace("num3 <= num2 <= num1" ) ;
```

EXERCISES:

1. Assume the following variables/values exist:

```
var num1:Number = 5 ;
var num2:Number = 10 ;
var num3:Number = 15 ;
var num4:Number = 20;
```

For each of the following state if the expression evaluates to true or false.

a. ((num1 < num2) || (num2 > num1))

b. ((num1 < num2) && (num2 > num1))

c. (((num1 < num2) && (num2 > num1)) || (num2 < num4))

d. (((num1 < num2) && (num2 < num3)) || (num4 < num2))

e. (((num1 < num2) && (num2 < num3)) && (num4 < num2))

2. Modify moving ball example (www.cs.du.edu/~leut/1671/FlashFiles/c3_leftRight5.fla) so that when the ball is moving left to right between 0 and (Stage.width / 2) the ball is increasing in size and when moving from left to right between (Stage.width / 2) and Stage.height it is decreasing in size. Then reverse the behavior right to left. Note, you can do this two ways: with nested if/else statement, or with compound if clauses that use the logical && operators.

   Here is an example of the behavior you should mimic: www.cs.du.edu/~leut/1671/FlashFiles/c3_lab1.html

3.5: Switch Statement

Sometimes in programming one wishes to see if a variable has a value equal to one of several different possible values. For example, perhaps a pet can be of 8 different types: {cat, dog, snake, fish, frog, hamster, gerbil, guinea pig}.

Perhaps one would write code to test for the type of pet as follows:

```
var pet:String = "ostrich"  ;

if (pet == "dog") trace("pet is a dog") ;
if (pet == "cat") trace("pet is a cat") ;
if (pet == "snake") trace("pet is a snake") ;
if (pet == "guinea pig") trace("pet is a guinea pig") ;
if (pet == "fish") trace("pet is a fish") ;
if (pet == "frog") trace("pet is a frog") ;
```

```
if (pet == "hamster") trace("pet is a hamster") ;
if (pet == "gerbil") trace("pet is a gerbil") ;

if ( (pet != "dog") && (pet != "cat") &&
  (pet != "snake") && (pet != "guinea pig") &&
  (pet != "fish") && (pet != "frog") &&
  (pet != "hamster") && (pet != "gerbil") )

  trace("pet is not one of the normal 8 pets") ;
```

This is correct, and does what we want, but is a tad inconvenient to say the least! ActionScript, and many other languages, give you a more convenient construct called the *switch* statement. The switch statement for the above would be:

```
switch (pet) {
        case "dog": trace("pet is a dog") ;
        break ;
        case "cat": trace("pet is a cat") ;
        break ;
        case "snake": trace("pet is a snake") ;
        break ;
        case "guinea pig": trace("pet is a guinea pig") ;
        break ;
        case "fish": trace("pet is a fish") ;
        break ;
        case "frog": trace("pet is a frog") ;
        break ;
        case "hamster": trace("pet is a hamster") ;
        break ;
        case "gerbil": trace("pet is a gerbil") ;
        break ;
        default: trace("pet is not one of the normal 8 pets") ;
}
```

Each possible value for the variable gets a “case” in the switch statement. The default code is executed when none of the cases match. Note, if more actions are desired for each case, curly-brace- delimited statement blocks could replace the trace statements in the switch structure. If more animal values are desired, it is easy to add the code, just add another case/break clause, and the default does not need to be changed.

3.6: Moving Gargoyles in Two Dimensions

Our examples up to now allow for a gargoyle to move back and forth, which I suppose if the gargoyle is tied to a building is okay, but these creatures have WINGS, they need to be able to move up and down.

First, lets remember the bounce left-right code. The following code found in:

www.cs.du.edu/~leut/1671/FlashFiles/c3_moveGarg1.fla

```
var garg1:MovieClip = attachMovie("gargoyle","garg1",1) ;
garg1._x = 300 ;
garg1._y = 300 ;
garg1._xscale = 20 ;
garg1._yscale = 20 ;
garg1._xscale *= -1 ;   // reverse direction of gargoyle image

var g1xv:Number = 5 ;


onEnterFrame = function () {

        // update x of  gargoyle
        garg1._x += g1xv ;


        // check gargoyle boundaries and reverse direction/scale if needed

        // see if going off right boundary, if so negate velocity
        if (garg1._x > Stage.width) {
                g1xv *= -1 ;    // reverse direction of gargoyle
                garg1._xscale *= -1 ;    // reverse direction of the image
        }

        // see if going off left boundary, if so negate velocity
        if (garg1._x < 0) {
                g1xv *= -1 ;    // reverse direction of gargoyle
                garg1._xscale *= -1 ;    // reverse direction of the image
        }
```

We have a variable named “g1xv” that holds the velocity of gargoyle1. When garg1._x becomes greater than Stage.width or less than zero, we negate this velocity to make the gargoyle move in the opposite directions. Also notice that we negate the MovieClip’s _xscale data member. This flips the orientation of the image so it is also moving face-forward.

We can create 2D movement by changing the MovieClip’s _y data member as well as its _x data member. Consider the following code:

www.cs.du.edu/~leut/1671/FlashFiles/c3_moveGarg2.fla

```
var garg1:MovieClip = attachMovie("gargoyle","garg1",1) ;
garg1._x = 300 ;
garg1._y = 300 ;
garg1._xscale = 20 ;
garg1._yscale = 20 ;
```

```
garg1._xscale *= -1 ;   // reverse direction of gargoyle image


var g1xv:Number = 5 ;
var g1yv:Number = 5 ;   // to hold the y-velocity


onEnterFrame = function () {

        // update x of  gargoyle
        garg1._x += g1xv ;
        garg1._y += g1yv ;   // update _y as well as _x



        // check gargoyle1 boundaries and reverse direction/scale if needed

        // see if going off right boundary, if so negate velocity
        if (garg1._x > Stage.width) {
                g1xv *= -1 ;    // reverse direction of gargoyle
                garg1._xscale *= -1 ;    // reverse direction of the image
        }

        // see if going off left boundary, if so negate velocity
        if (garg1._x < 0) {
                g1xv *= -1 ;    // reverse direction of gargoyle
                garg1._xscale *= -1 ;    // reverse direction of the image
        }

        // see if going off bottom boundary, if so negate y-velocity
        if (garg1._y > Stage.height) {
                g1yv *= -1 ;    // reverse direction of gargoyle
        }

        // see if going off top boundary, if so negate y-velocity
        if (garg1._y < 0) {
                g1yv *= -1 ;    // reverse direction of gargoyle
        }
}
```

The new code here is the added second velocity, g1yv, for storing the y-velocity and on every frame entry we update the gargoyle's _y location as well as its _x location. Also, we now need to check that the gargoyle does not go off the screen out the top or bottom, so we need to add those checks also, and negate the y-velocity if the _y coordinate value becomes less than zero or greater than Stage.height.

3.6: Creating More Fear: Many Gargoyles

What is more terrifying than a gargoyle? Two gargoyles of course. The easy way (or is it?) to create two gargoyles is to have two MovieClips with different names, and two sets of {x,y}-velocities, one set for each gargoyle. Then, on every frame entry we need to

update the {_x , _y } data members of both gargoyles, and we need to check the four boundaries for each gargoyle. The code for this:

www.cs.du.edu/~leut/1671/FlashFiles/c3_moveGarg3.fla

```
var garg1:MovieClip = attachMovie("gargoyle","garg1",1) ;
garg1._x = 300 ;
garg1._y = 300 ;
garg1._xscale = 20 ;
garg1._yscale = 20 ;
garg1._xscale *= -1 ;   // reverse direction of gargoyle image

// make a second gargoyle
var garg2:MovieClip = attachMovie("gargoyle","garg2",2) ;
garg2._x = 100 ;
garg2._y = 100 ;
garg2._xscale = 20 ;
garg2._yscale = 20 ;
garg2._xscale *= -1 ;   // reverse direction of gargoyle image


var g1xv:Number = 5 ;
var g1yv:Number = 5 ;
var g2xv:Number = 2.5 ;
var g2yv:Number = 2.5 ;

onEnterFrame = function () {

        // update x,y of both gargoyles
        garg1._x += g1xv ;
        garg1._y += g1yv ;
        garg2._x += g2xv ;
        garg2._y += g2yv ;

        // check gargoyle1 boundaries and reverse direction/scale if needed

        // see if going off right boundary, if so negate velocity
        if (garg1._x > Stage.width) {
                g1xv *= -1 ;    // reverse direction of gargoyle
                garg1._xscale *= -1 ;    // reverse direction of the image
        }

        // see if going off left boundary, if so negate velocity
        if (garg1._x < 0) {
                g1xv *= -1 ;    // reverse direction of gargoyle
                garg1._xscale *= -1 ;    // reverse direction of the image
        }

        // see if going off bottom boundary, if so negate velocity
        if (garg1._y > Stage.height) {
                g1yv *= -1 ;    // reverse y-direction of gargoyle
        }
```

```
        // see if going off top boundary, if so negate velocity
        if (garg1._y < 0) {
                g1yv *= -1 ;    // reverse direction of gargoyle
        }

        // check gargoyle2 boundaries and reverse direction/scale if needed

        // see if going off right boundary, if so negate velocity
        if (garg2._x > Stage.width) {
                g2xv *= -1 ;    // reverse direction of gargoyle
                garg2._xscale *= -1 ;    // reverse direction of the image
        }

        // see if going off left boundary, if so negate velocity
        if (garg2._x < 0) {
                g2xv *= -1 ;    // reverse direction of gargoyle
                garg2._xscale *= -1 ;    // reverse direction of the image
        }

        // see if going off bottom boundary, if so negate velocity
        if (garg2._y > Stage.height) {
                g2yv *= -1 ;    // reverse y-direction of gargoyle
        }

        // see if going off top boundary, if so negate velocity
        if (garg2._y < 0) {
                g2yv *= -1 ;    // reverse direction of gargoyle
        }

}
```

If you want the gargoyles to move at different speeds, just play with the two sets of velocities, g1xv, g1yv, g2xv, and g2yv.

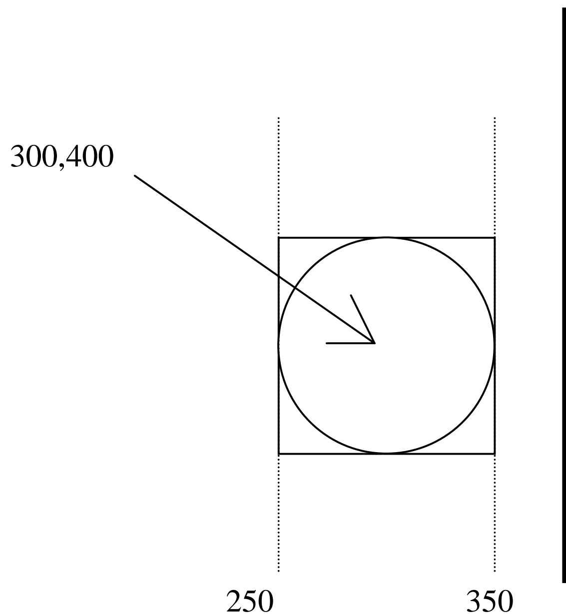
So, you want more fear in your life? How about 3 gargoyles? 4? 5? 10? Can you imagine typing all the code to make 10 gargoyles move hither and yon? There HAS to be a better way, and, as we will show you in Chapter ZZZ, there is a much better way.

3.7: Respect Those Boundaries

You pushed and pushed your friend until s/he had to set some boundaries. We are human, we learn, and hopefully we speak up when we need to. But flash won't complain, it is up to you to make sure you respect boundaries. Did you notice that that gargoyle in the last three examples sort of went part way off the screen before bouncing? Perhaps you really want them to bounce the moment they touch the boundary? The key to respecting boundaries is to understand how flash stores images in MovieClips and how *registration* works.

When you create a MovieClip in Flash by using the Modify->Convert To Symbol dialogue, the _width and _height members are set to the width and the height of the

image you are converting. You can adjust these by changing _xscale and _yscale. To calculate these values internally Flash encloses your drawing with the smallest upright (i.e. orthogonal to the axes) rectangle to completely enclose the image. In addition to _width and _height, there is the concept of *registration.* As we know, a MovieClip has a (_x , _y) pair to designate a location. But exactly how does this relate to the image? During the Modify->Convert To Symbol dialogue, you can see an entry called registration. By default yours is set to the top left corner, but you can change this to any of 8 other locations. In my examples the middle was chosen.
Say a MovieClip, holding a circle, has a width of 100 pixels, a height of 100 pixels, and is currently located at (300 , 400). If the registration is set to the center, this means that the image actually spans from 250 to 350 along the x-coordinate, and from 350 to 450 along the y-coordinate.



The above example shows a MovieClip lets name it MC1, with a circle image, with a width and height of 100, with a central registration point centered at location 300,400. If this MovieClip is moving towards a barrier at x-coordinate 375, represented as a black line above, then the following boundary check will NOT work:

```
if (MC1._x > 375) {
        // do something, like negate an x-velocity
}
```

The reason is that the MovieClip will actually move half way across the boundary before the central registration point crosses the boundary. Instead, we need to add on half of the MovieClip's width when doing the check as follows:

```
if (  (MC1._x  + ( MC1._width/2)  )  > 375) {
```

```
        // do something, like negate an x-velocity
}
```

This would do what we want. In this spirit, check out the following modification to our gargoyle code (even gargoyles should respect boundaries):

www.cs.du.edu/~leut/1671/FlashFiles/c3_moveGarg4.fla

```
var garg1:MovieClip = attachMovie("gargoyle","garg1",1) ;
garg1._x = 300 ;
garg1._y = 300 ;
garg1._xscale = 20 ;
garg1._yscale = 20 ;
garg1._xscale *= -1 ;   // reverse direction of gargoyle image


var g1xv:Number = 5 ;
var g1yv:Number = 5 ;


onEnterFrame = function () {

        // update x of  gargoyle
        garg1._x += g1xv ;
        garg1._y += g1yv ;


        // see if going off right boundary, if so negate velocity
        if ( (garg1._x + (garg1._width/2) ) > Stage.width) {
                g1xv *= -1 ;    // reverse direction of gargoyle
                garg1._xscale *= -1 ;    // reverse direction of the image
        }

        // see if going off left boundary, if so negate velocity
        if ( (garg1._x - (garg1._width/2)) < 0 ) {
                g1xv *= -1 ;    // reverse direction of gargoyle
                garg1._xscale *= -1 ;    // reverse direction of the image
        }

        // see if going off bottom boundary, if so negate y-velocity
        if ( (garg1._y + (garg1._height/2))  > Stage.height) {
                g1yv *= -1 ;    // reverse direction of gargoyle
        }

        // see if going off top boundary, if so negate y-velocity
        if ( (garg1._y - (garg1._height/2) ) < 0) {
                g1yv *= -1 ;    // reverse direction of gargoyle
        }

}
```

Notice that all the boundary checks now add or subtract half of the width or height.

What would happen if you set the registration point to the top corner? Then your checks for the left side or top side would be correct, but you need to add in the full width and height when checking the bottom and right boundaries.

3.7: How Fast Do Gargoyles Move?

That depends on how fast you tell them to move. The speed is determined by the distance the gargoyle is moved each frame, as represented in our code the velocity, and the number of frames per second played by Flash. Flash allows you to change the number of frames per second. The default parameter value is 12 frames per second, my examples have set the frame rate to 24 FPS. You can change this in the Properties panel of the main document. Click someplace outside the stage to get the main properties panel. About in the center-top you will see a frame rate box. Go ahead and change this to 12 or 36 and re-run (ctrl-enter) the animation.

Note, if you change the frame rate to 12, and also change the velocity to be double the current values, then the effective rate is the same, but the animation looks jerky. On the other hand, if you set the frame rate too high and the animation is run on a slower computer, then the computer will not be able to execute all the code. In this case Flash just starts dropping frames! This can be quite a problem, for example you might have a frame that deals with the collision reaction from two balls hitting each other. If that frame is dropped the balls might just go right through each other instead of bouncing because the “bounce” code is never executed.

So, what is the right frame rate? There is no easy answer, but the default that I see on the web for games seems to be 24 frames per second. So, I suggest we all use 24 fps. Remember, you have no control over the speed of the computer that someone uses to view your animation/game, hence you need to design for lame machines, not the latest and greatest muscle PC.