Chapter 4: Collisions and Mouse Interactivity

We now know quit a bit: how to use objects, the difference between a class and an object, variables, if/else, Boolean expressions, event driving programming (the onEnterFrame function), and how to bounce MovieClip objects around. But we hardly know how to create games. Games require interactivity of some sort, most commonly using the mouse or keyboard. In this chapter we will learn a bit of mouse interactivity, but first, we learn the concept of collision.

4.1: Collision Detection

Say you have created the latest "Tom and Jerry" game, and you manage to get Tom and Jerry to bump into each other. You need a way to make something happen (like Tom politely offering Jerry some tea and scones?). But first, you need a way to detect that two objects have in fact hit each other. Flash provides a built in "hit test" for MovieClips. Remember that MovieClip is a class and that classes have data members and methods. One built in MovieClip method is *hitTest()*. There are several ways to use the *hitTest()* method.

First, to create something to hit each other, lets create two bouncing gargoyles:

www.cs.du.edu/~leut/1671/FlashFiles/c4_collision1.fla

Nothing new here, just two gargoyles bouncing correctly and respecting those boundaries.

Now lets modify the code to detect when the two gargoyles bump into each other and when they do stop movement. Load the code and run it:

www.cs.du.edu/~leut/1671/flashFiles/c4_collision2.fla

The code is identical to the previous example except the following lines of code are added inside the onEnterFrame function:

```
if ( garg1.hitTest(garg2) ) {  // then set the velocities to zero
        trace("they intersect") ;
        g1xv = 0 ;
```

```
            g1yv = 0
            g2xv = 0 ;
            g2yv = 0 ;
}
```

The first line tests to see if the two gargoyles are touching. This is done using the *hitTest()* method of the MovieClip class. The method takes an *argument*, where the argument here is another instance of a MovieClip. The function returns true if the two MovieClips intersect. Actually, it returns true if the bounding boxes of the two images intersect. The last four lines are executed if *hitTest()* returns true, and the 4 lines simply set the x and y-velocities of both gargoyles to zero. Later we will write our own hitTest code, but for now it is easier to use the code built into the class.

When you run this code the trace statement keeps printing "they intersect" over and over. The reason is that even though you set the velocities to zero, the onEnterFrame function is entered at the Frames Per Second rate. Each time the onEnterFrame function is entered the gargoyles intersect, since we stopped them while intersecting, and hence print out the trace statement again and again. One way to stop this is to cancel the onEnterFrame function. This can be done as follows:

www.cs.du.edu/~leut/1671/flashFiles/c4_collision2b.fla

```
if ( garg1.hitTest(garg2) ) {
        trace("They intersect") ;
        g1xv = 0 ;
        g1yv = 0
        g2xv = 0 ;
        g2yv = 0 ;
        onEnterFrame = {} ;
}
```

The last line is the only difference from the preceding example. The line redefines the function *onEnterFrame* to be nothing, hence, it is no longer called and the first trace statement is only called once.

Exercises:

1. Modify the bouncing gargoyle example so that when the gargoyles hit one another they bounce away from each other.
2. Modify again so that when the gargoyles hit they not only bounce away but also start going faster.

4.2: Mouse Tracking

Time to add interactivity! One way to make it interactive is to use the mouse. Flash provides one really easy way to use the mouse. Like a MovieClip object, the mouse object also has x and y locations, denoted _xmouse and _ymouse. What is really cool, is that you can access these parameters from anyplace inside ActionScript code. Load and run the following code:

www.cs.du.edu/~leut/1671/flashFiles/c4_mouse1.fla

When you run the code, just move the mouse over a gargoyle and the gargoyle stops. Move to the other gargoyle to stop it. This cool trick is accomplished by adding the following lines to the onEnterFrame function:

```
// check if mouse over garg1, if so stop it
if ( garg1.hitTest( _xmouse, _ymouse) ) {
        g1xv = 0 ;
        g1yv = 0 ;
}

// check if mouse over garg2, if so stop it
if ( garg2.hitTest( _xmouse, _ymouse) ) {
        g2xv = 0 ;
        g2yv = 0 ;
}
```

This is another way to use the *hitTest()* method. This is a common technique in object-oriented languages: the same method can be used different ways. The methods differ in that they take different arguments. In the previous example of *hitTest()* used in section 4.1, the function took one argument: another MovieClip. In this usage the method takes two arguments: an x-coordinate and a y-coordinate. The methods tests to see if the point passed in is contained in the bounding box of the MovieClip and if so returns true, else false. Since _xmouse and _ymouse together specify the current location of the mouse, this test returns true if the current mouse x,y location is inside the MovieClip's bounding box. If so, the velocities of that MovieClip is set to zero.

I am sure that by now you have noticed the mouse location is represented on the screen as a little area. But would an arrow really stop a gargoyle? I don't think so, but love might. Download and run the next example:

www.cs.du.edu/~leut/1671/flashFiles/c4_mouse2.fla

Here you see the mouse pointer has been replaced by a heart. Move the love (heart) to a gargoyle and it stops. This is done by creating a heart shaped movie clip and then creating an instance of it on the stage:

```
var heart:MovieClip = attachMovie("heart","h3",3) ;
heart._x = _xmouse ;
heart._y = _ymouse ;
heart._xscale = 20 ;
heart._yscale = 20 ;

Mouse.hide() ;
```

Unless you hide the mouse arrow it will show up in addition to the heart. Hiding the mouse arrow is accomplished with the statement: *Mouse.hide( ) ;* Next, the location of this MovieClip instance needs to be updated to "stick" to the mouse. In the onEnterFrame function we have:

```
// update the heart location to keep it on the mouse
heart._x = _xmouse ;
heart._y = _ymouse ;
```

Finally, it would be great if there is a way to restart the gargoyles once they have been stopped. One way to do this is found in:

www.cs.du.edu/~leut/1671/flashFiles/c4_mouse3.fla

Here a text box with the word "restart" was created and then converted into a MovieClip. An instance of the MovieClip is attached:

```
// Put a restart object on the stage
var restart:MovieClip = attachMovie("restartClip","r4",4) ;
restart._x = 50 ;
restart._y = 100 ;
```

Now, we just need to check to see if the mouse is inside the restart box by using the same hitTest() function as the gargoyles, and, if it is, just reset the velocities to the original values:

```
// check if mouse restart, if so restart both gargoyles if stopped
if ( restart.hitTest( _xmouse, _ymouse) ) {
        g1xv = 5 ;
        g1xv = 5 ;
```

```
            g2xv = 5 ;
            g2yv = 3 ;
}
```

Exercises:

1. Modify example c4_mouse2.fla so that when you tag a gargoyle with a mouse it causes that gargoyle to speedup.

4.3: Are we there yet? Are we there yet? Are we there yet? Or Mouse Handlers 101

Have you ever gone on a long family car trip. These were really great in the summer before air-conditioning. Did you enjoy harassing your parents with frequent queries of "Are we there yet?". Some day you may have kids of your own and experience this pleasure from the other side. In the meantime, we are doing exactly the same thing with the way we are checking for mouse collisions in flash. Every single time we enter a frame we check to see if the current mouse location intersects with the gargoyles, even if we have not moved the mouse! This is the equivalent asking the driver every mile "are we there yet?" A much better approach is to say, "Honored parental unit, please tell me when we get there", and then just sit back and do something productive like intimidating a sibling. Likewise, for actionscript, how about if we just tell the built in Mouse class to let us know when the mouse has been moved or clicked rather than checking every frame! We do this with something called a *listener*. A listener is just a special event handling function that listens for an action to occur and then handles it when the event occurs. It is another example of "event driven programming", like onEnterFrame, except here the event is a mouse action.

If you look at the code in www.cs.du.edu/~leut/flashFiles/mouse3.fla, you see that on EVERY frame entry we check to see if the current mouse location intersects with the gargoyles or the restart area. We do this check every frame even if we do not move the mouse! This is pretty much like a kid asking "are we there yet" without even looking outside the window. It makes more sense, and requires less computation, to wait for the mouse to move and then check to see if the mouse intersects with an object.

The way this is done in ActionScript is to create a mouse handler. An "event handler" is a function that is called after an event occurs. In this case the event would be a mouse action, in particular a mouse movement:

www.cs.du.edu/~leut/1671/flashFiles/mouse3b.fla

```
mouseHandler = new Object() ;
mouseHandler.onMouseMove = function() {

        // check if mouse over garg1, if so stop it
        if ( garg1.hitTest( _xmouse, _ymouse) ) {
                g1xv = 0 ;
                g1yv = 0 ;
        }

        // check if mouse over garg2, if so stop it
        if ( garg2.hitTest( _xmouse, _ymouse) ) {
                g2xv = 0 ;
                g2yv = 0 ;
        }

        // check if mouse restart, if so restart both gargoyles if stopped
        if ( restart.hitTest( _xmouse, _ymouse) ) {
                g1xv = 5 ;
                g1xv = 5 ;
                g2xv = 5 ;
                g2yv = 3 ;
        }

}

Mouse.addListener(mouseHandler) ;
```

The first line of the above code creates a new object named “mouseHandler”. In ActionScript we can create an arbitrary object without specifying its data members and methods and then later add them. In general this is very bad programming practice. It is a common way to do things in ActionScript due to how the language evolved. In general we will not follow this technique for creating objects, with the exception of creating mouse and keyboard handlers, for which we will follow this practice.

Next, we define a function *onMouseMove* and attach it to the mouseHandler object. The code inside the *onMouseMove* function is executed whenever the user clicks the mouse down. In the example above, the code checks to see if the current _xmouse and _ymouse coordinates intersect with gargoyle1, gargoyle2, or the restart MovieClips. If there is a gargoyle hit, the corresponding velocities are set to zero. If there is a hit with restart, both sets of velocities are reset.

The last line adds a listener to our mouseHandler object. A listener does exactly what the name applies, it “listens”. The mouse itself is an object, and the addListener method allows one to associate an object to listen to any changes the mouse makes. The object in this case is the mouseHandler that we just defined. There are three events a listener can listen for: onMouseDown, onMouseUp, and onMouseMove.

This leads right into a more common game use of a mouse: react when the player clicks on an object. Lets say we want to change our animation so that clicking on a gargoyle results in stopping movement of the gargoyle:

www.cs.du.edu/~leut/1671/flashFiles/c4_mouse4.fla

The code here is identical to the preceding example except we change the mouseHandler function's name from *onMouseMove to onMouseDown*. Now, the mouseHandler code is only executed when the mouse button is pressed down. Slick, no?

Perhaps you think stopping gargoyles with hearts is just to darn weird. Perhaps you would prefer something more "traditional", say a scope from a rifle. Download and run:

www.cs.du.edu/~leut/1671/flashFiles/c4_mouse5.fla

This code is identical to the previous example, only when we attach object to serve as the mouse pointer we attach a riflescope. To see this just hit ctrl-L to open the library. Note, I left the variable named "heart" even though it is now a riflescope. Lazy!

Perhaps we want the gargoyle to disappear when we hit it. The can be done be removing the MovieClip from the variable:

www.cs.du.edu/~leut/1671/flashFiles/c4_mouse6.fla

```
// check if mouse over garg1, if so stop it
if ( garg1.hitTest( _xmouse, _ymouse) ) {
        garg1.removeMovieClip() ;
}

// check if mouse over garg2, if so stop it
if ( garg2.hitTest( _xmouse, _ymouse) ) {
        garg2.removeMovieClip() ;
}
```

The rest of the code is identical to the previous example. Here, when the user clicks on the mouse, the hitTest is run, and if there is a hit, the MovieClip is removed from the gargoyle1 or gargoyle2 variable using the removeMovieClip() command.

Now lets say you want to change things so that when you hit the gargoyles they turn into hearts. To do this, just remove the gargoyle and then attach a heart:

www.cs.du.edu/~leut/1671/flashFiles/c4_mouse7.fla

```
	mouseHandler.onMouseDown = function() {

	// check if mouse over garg1, if so stop it
	if ( garg1.hitTest( _xmouse, _ymouse) ) {
		garg1.removeMovieClip() ;
		garg1 = attachMovie("heart","garg1",1) ;
	}

	// check if mouse over garg2, if so stop it
	if ( garg2.hitTest( _xmouse, _ymouse) ) {
		garg2.removeMovieClip() ;
		garg2 = attachMovie("heart","garg2",2) ;
	}

}
```

When you run this and click on a gargoyle it disappears and is replaced by a heart as expected, but the heart is HUGE and it is over in the top corner instead of where you clicked on the gargoyle. The reason is that when you remove a MovieClip object you lose all the member values of that object. So when we then attach the heart object, it's _x, _y, _width or _height member values are all initialized with the default, which for _x and _y is 0, and for _width and _height are the values of the object in the library. If you want the new object to have the same member values as the removed MovieClip object you need to store the member values in temporary variables before removing the object and then assign the values to the new object. This is done for garg1 in:

www.cs.du.edu/~leut/1671/FlashFiles/c4_mouse8.fla

The key code is:

```
	if ( garg1.hitTest( _xmouse, _ymouse) ) {

		var tempX:Number = garg1._x ;
		var tempY:Number = garg1._y ;
		var tempWidth:Number = garg1._width ;
		var tempHeight:Number = garg1._height ;
		garg1.removeMovieClip() ;
		garg1 = attachMovie("heart","garg1",1) ;
		garg1._x = tempX ;
		garg1._y = tempY ;
		garg1._width = tempWidth ;
		garg1._height = tempHeight ;

	}
```

The variables tempX, tempY, tempWidth, and tempHeight hold the values. When you click on the gargoyle stored in variable garg1 you will see it is replaced with a new hear object of the same size and in the same place. This technique of storing values in temporary variables is commonly uses in computer programming.

As our final example of having fun with mice, lets use the mouse to push a gargoyle around the screen. First download and run:

www.cs.du.edu/~leut/1671/flashFiles/c4_mouse9.fla

Just move the mouse to "bump" the gargoyle around. When you bump the gargoyle from the right, the gargoyle moves to the left. From the left causes a movement to the right, top to the bottom, bottom to the top. The code to make this happen is in the mouseHandler:

```
mouseHandler = new Object() ;
mouseHandler.onMouseMove = function() {

        // check if mouse (ball) intersects gargoyle
        if ( garg1.hitTest( heart ) ) {
                if (ball._x < garg1._x)
                        g1xv += 0.1 ;
                if (ball._x > garg1._x)
                        g1xv -= 0.1 ;
                if (ball._y < garg1._y)
                        g1yv += 0.1 ;
                if (ball._y > garg1._y)
                        g1yv -= 0.1 ;
        }

}

Mouse.addListener(mouseHandler) ;
```

If the mouse moves, we check to see if it hits the gargoyle. If so, we check to see if the heart attached to the mouse is to the right of the gargoyle, if so we decrease the x-velocity. Similar checks and adjustments are made for left, above, and below.