9

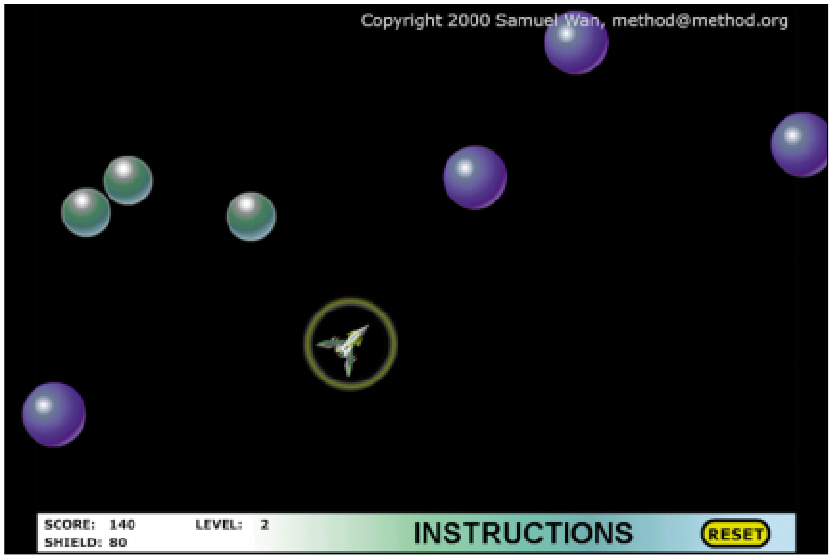
PROGRAMMING THE COLLISION DETECTION OF MULTIPLE PROJECTILES

By Samuel Wan

The concept of collision detection is easy enough to understand: Compare the areas of two movie clips to see if they overlap. The **hitTest** function in ActionScript makes the detection of colliding movie clips simple enough by comparing two objects to see if they overlap. If they do overlap at some point, the two movie clips are considered to have collided against one another. The **hitTest()** function in Actionscript is a very convenient method of collision detection. This function compares the bounding boxes of two movie clips to see if they overlap. Bounding boxes are like imaginary rectangles around the edges of a movie clip that define its left edge, right edge, top edge, and bottom edge. For example, if you were to compare the collision of two movie clips, bubble and ship, you would simply call the **hitTest** method of one movie clip and insert the instance name of the other movie clip as the parameter. The method returns a true or false: true if a collision was detected, or false if no collision was detected.

```
IsCollided = Bubble.hitTest("ship");
```

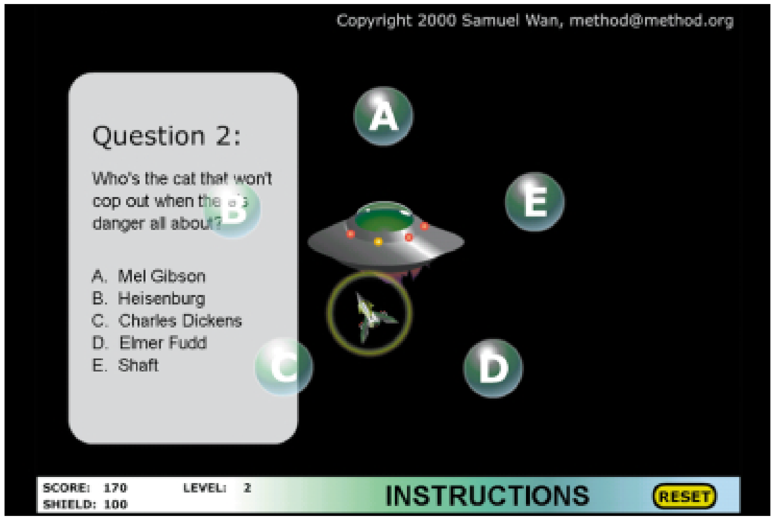
In the preceding line, the bubble movie clip's hitTest method is called to compare against the ship movie clip. If the bubble movie clip



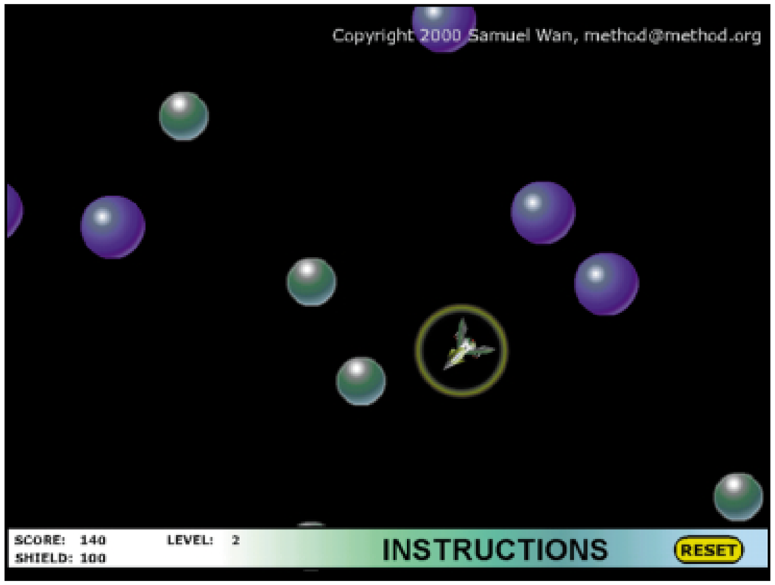
overlaps with the ship movie clip, the **hitTest** function will return a **true** to the ***IsCollided*** variable. Otherwise, the function will return a **false** if they don't overlap. This line of code is simple enough, but unlike the ship module, which contains only one ship, the bubble module contains many bubbles. How do you use the **hitTest** function to detect collision of the ship against more than one bubble? Objects in constant motion, also referred to as projectiles, change their position after every iteration of ActionScript. You could take a very crude approach and write code to compare the collision of each bubble to the ship like so:

```
IsCollided = Bubble1.hitTest("ship");
IsCollided = Bubble2.hitTest("ship");
IsCollided = Bubble3.hitTest("ship");
IsCollided = Bubble4.hitTest("ship");
```

However, writing out a line of code for every single projectile in the game is not only an inefficient way to program, but it's also tough on your fingers. So the question remains: How do you apply collision detection to multiple projectiles in an efficient, elegant way? To answer this question, you need to come up with a way to associate all the bubbles into a single, easy-to-use catalog system.



This scene has several bubbles that must be able to detect a collision with the ship.



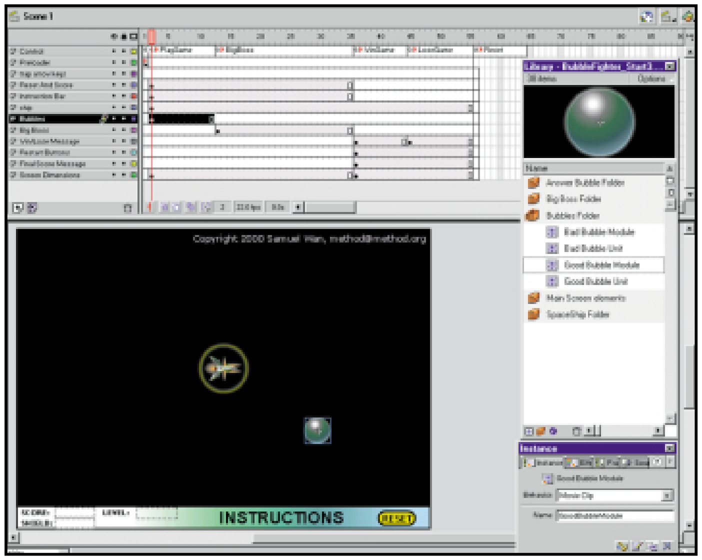
Multiple copies of the same two bubbles must be able to detect a collision with the ship in this scene.

Preparing for Associative Arrays

A catalog system is a handy way to reference objects—like the drawers of alphabetically organized cards that provide information about books in a library. In the same way, associative arrays allow us to build a small box of cards that are "associated" with objects, such as movie clips. If an associative array is like a box of reference cards, the card inside the box would be the equivalent of an "element" inside the associative array. This element could refer to an object, a variable, a value, and so on, but in this case, you want to build an array and associate its elements to movie clips. Let's walk through the construction of an associative array of bubbles in the game.

1 Open the BubbleFighter_Start3.fla file in the Chapter 09 folder and save it to your local hard drive. (The final file is BubbleFighter_Final3.fla.)

2 In the main timeline, look for the layer named Bubbles and select the second frame (frame 2) on that layer. Open the Bubbles Folder in the Library and drag the Good Bubble Module movie clip onto the Stage.


Drag the GoodBubbleModule movie clip onto the Stage at the second frame of the Bubbles layer.

3 Give the movie clip an instance name of **GoodBubbleModule**, and double-click on it to take a quick peek inside.

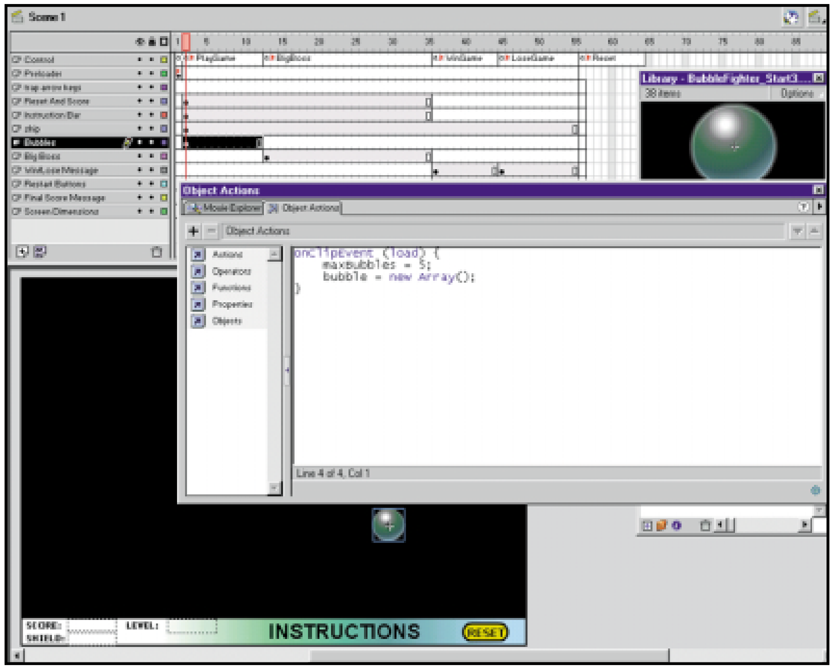
You'll notice that the movie clip contains another movie clip with an instance name called originalbubble. This movie clip contains the graphics that draw a bubble, and it acts as a template from which you can duplicate more bubbles.

4 Go back to the original timeline and open up the Actions panel of the GoodBubbleModule movie clip. Apply this ActionScript to initialize the bubble module:

```
onClipEvent (load) {
    maxBubbles = 5;
    bubble = new Array( );
}
```

Apply the **onClipEvent (load)** code to the GoodBubbleModule movie clip.

Note: As explained in previous chapters of this arcade game, **onClipEvent(load)** is an event handler that will execute all the code within its brackets one time after the movie clip has loaded.



```
onClipEvent (load) {
    maxBubbles = 5;
    bubble = new Array();
}
```

The Object Actions window shows the **onClipEvent(load)** code assigned to the GoodBubbleModule movie clip.

You created one variable and one array as soon as the GoodBubbles movie clip was loaded into the Flash player. The variable ***maxBubbles*** stores the maximum number of bubbles needed for the GoodBubbles module. The **bubbles** array will act as the cataloging box to keep track of all the bubbles.

Duplicating Bubbles and Associating Them to an Array

Now that you've initialized this module, you're ready to work with the bubble array. In this section, you add the ActionScript to accomplish three tasks. The first task is to duplicate the original Bubble movie clip inside the GoodBubbleModule movie clip into a new bubble movie clip with an appended number (such as bubble0, bubble1, bubble2, bubble3, and so on). The second task is to associate an element of the **bubbles** array to each new bubble instance. The third task is to assign random speed, direction, and original position of the new bubbles.

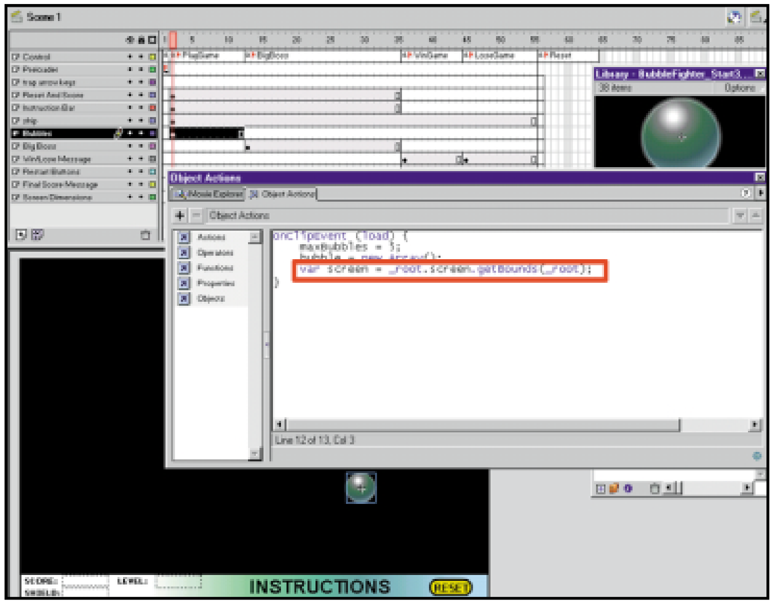
1 After the code that creates the new bubble array, insert this code:

```
var screen = _root.screen.getBounds(_root);
```

Add the ***screen*** variable to the GoodBubbleModule movie clip.

This code creates an object called screen that holds the left, right, upper, and lower bounds of the game area on the main timeline. Thus, the screen object contains properties of ***screen.xMin***, ***screen.xMax***, ***screen.yMin***, and ***screen.yMax***.

Note: See the previous chapter for more information about bounds and how you use the screen movie clip to get the bounds of the game area.



```
onClipEvent (load) {
    maxBubbles = 5;
    bubble = new Array();
    var screen = _root.screen.getBounds(_root);
}
```

The ***screen*** variable is added to the GoodBubbleModule movie clip.

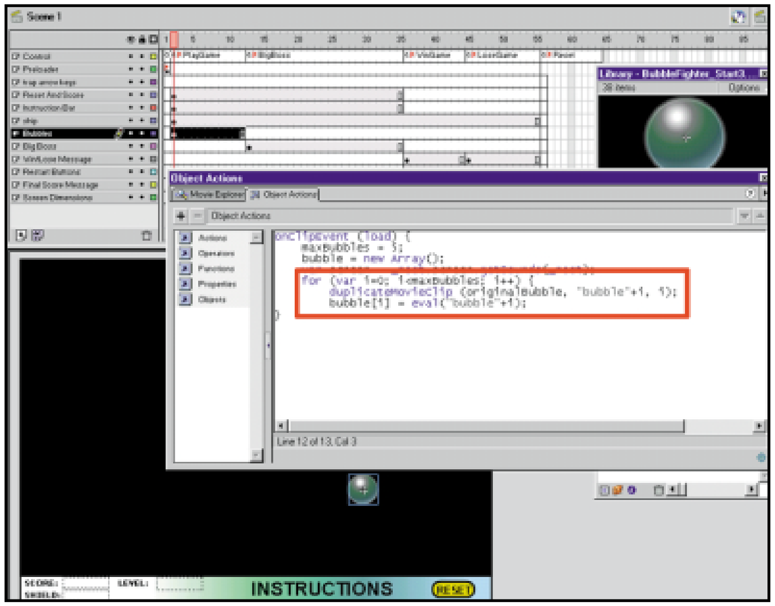
2 Use this code to duplicate to create a new instance of the original Bubble movie clip, and then assign that instance to the **bubble[]** array:

To do this many times without writing many lines of code, you use a **for** loop to run through the whole process as many times as specified by the value of **maxBubbles**. The first line of code in the **load** event handler gave **maxBubbles** a value of **5**, so you loop through the duplication and array assignment five times.

Note that the variable ***i*** will increment from 0 through 4, for a total of five times (counting 0, 1, 2, 3, and 4). This code creates a new instance with the name bubble followed by the value of ***i***, so that you create four duplicated movie instances with the names bubble0, bubble1, bubble2, bubble3, and bubble4, at depths from 0 through 4, respectively.

```
for (var i = 0; i < maxBubbles; i ++){
    duplicateMovieClip (originalBubble, "bubble" + i, i);
    bubble[i] = eval("bubble" + i);
```

Continue adding to the **onClipEvent(load)** code on the GoodBubbleModule movie clip.



```
onClipEvent (load) {
    maxBubbles = 5;
    bubble = new Array();
    var screen = _root.screen.getBounds(_root);
    for (var i=0; i<maxBubbles; i++) {
        duplicateMovieClip (originalBubble, "bubble"+i, i);
        bubble[i] = eval("bubble"+i);
}
```

The **for** loop is added to the GoodBubbleModule movie clip.

3 Set the bubble's location to a random *x* and *y* position, within the *x* and *y* bounds of the screen movie clip, using this code:

```
bubble[i]._x = random(screen.xMax);
bubble[i]._y = random(screen.yMax);
```

Use this code to set a random location for each of the duplicate bubbles.

If all the bubbles moved at the same speed in the same direction, the game would feel too artificial, so you want to assign two variables, one for horizontal ***xspeed*** and one for vertical ***yspeed***, inside each bubble's movie clip.

4 Use this code to assign the variables:

```
        bubble[i].xspeed = (int(random(5)) + 2) * (1 - (random(2)* 2));
        bubble[i].yspeed = (int(random(5)) + 2) * (1 - (random(2)* 2));
    }
```

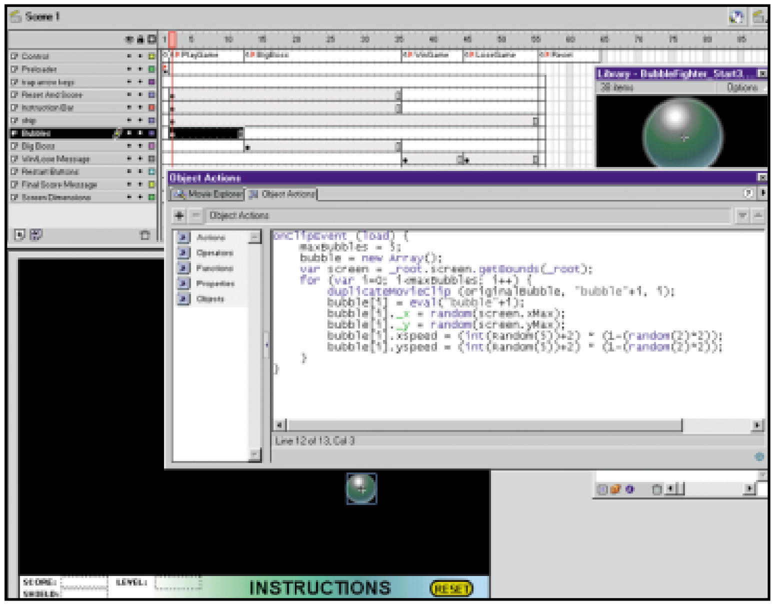
Insert this code to set a random speed for each of the duplicate bubbles.

The formula in the two lines actually works in two parts to generate a random value for the ***xspeed*** and ***yspeed*** directions. The first part of the formula, **(int(random(5) + 2))**, generates a random number from 2 through 6.

The second part of this formula, **(1 - (random(2)* 2))**, returns a value of either **1** or **–1**. Its actual function is a bit more complicated than the first part, but it's much more worthy of a closer look. Note that the expression **random(2)** will return either zero or one (**0** or **1**). When you multiply that random expression by 2 to express **(random(2) * 2)**, you receive a value of either zero or 2 (**0** or **2**). The results are limited to these two values because zero times two is still zero (0 * 2 = 0), whereas one times two is equal to two (1 * 2 = 2). Subtracting this random expression of either zero or 2 from a number of 1 will produce only two possible calculations:

(1 - (random(2)* 2))
First possible calculation: 1 – 0 = 1
Second possible calculation: 1 – 2 = -1

Multiplying both parts of the formula results in random values from **–6** through **–2** or **2** through **6**. This allows us to tell the bubble to move randomly at a speed ranging from two through six in either a forward or backward direction. Because you use the same formula to set the values for ***xspeed*** and

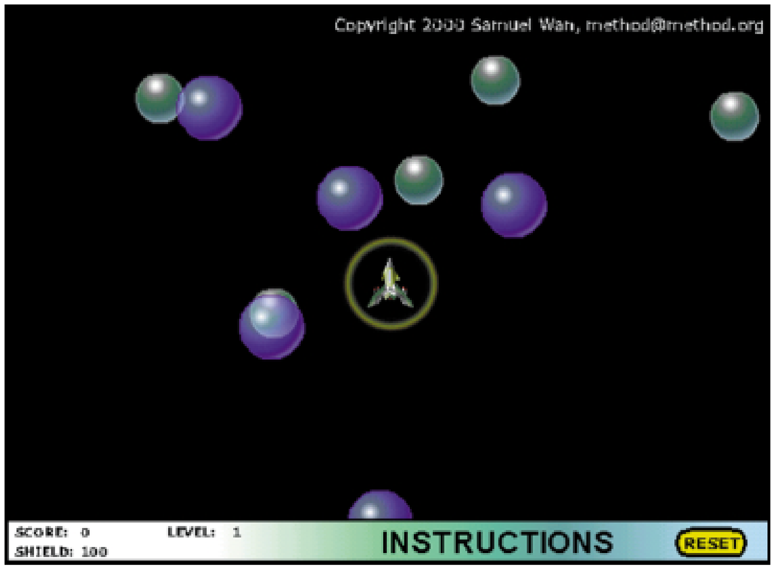


```
onClipEvent (load) {
    maxBubbles = 5;
    bubble = new Array();
    var screen = _root.screen.getBounds(_root);
    for (var i=0; i<maxBubbles; i++) {
        duplicateMovieClip (originalBubble, "bubble"+i, i);
        bubble[i] = eval("bubble"+i);
        bubble[i]._x = random(screen.xMax);
        bubble[i]._y = random(screen.yMax);
        bubble[i].xspeed = (int(Random(5))+2) * (1-(random(2)*2));
        bubble[i].yspeed = (int(Random(5))+2) * (1-(random(2)*2));
    }
}
```

The additional code is added to the **onClipEvent(load)** code.

yspeed, those two lines of code will make the bubble move forward or backward, upward or downward, within a specific range of random numbers.

After all the duplication, positioning, and randomizing of speed has been completed, you no longer need the originalBubble movie clip, so you simply make it invisible.



The bubbles appear randomly on the stage and move in random directions and at random speeds.

Note: For more detailed explanations about the mechanics inside the **random** function, refer to the Macromedia Flash documentation under the keyword "Random".

5 Use this code to make the movie clip invisible and close out the **onClipEvent(load)** event handler with a closing bracket:

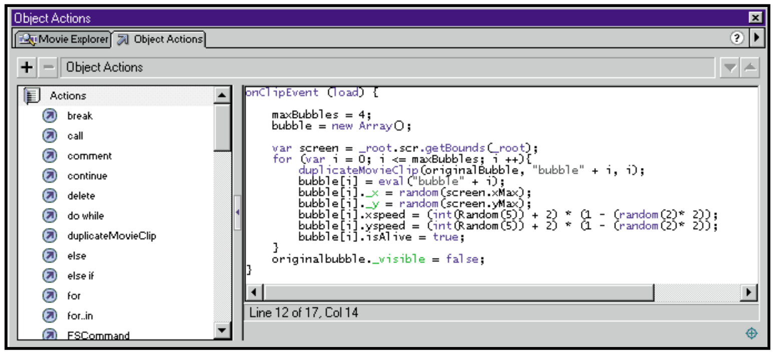
```
        originalBubble. _visible = false;
}
```

Insert the code that makes the originalBubble movie clip invisible.

Your final code for the **onClipEvent(load)** event handler for the goodbubble module should look like this:

```
onClipEvent (load) {
    maxBubbles = 5;
    bubble = new Array();
    var screen = _root.screen.getBounds(_root);
    for (var i = 0; i < maxBubbles; i ++){
        duplicateMovieClip (originalBubble, "bubble" +
        ➥i, i);
        bubble[i] = eval("bubble" + i);
        bubble[i]._x = random(screen.xMax);
        bubble[i]._y = random(screen.yMax);
        bubble[i].xspeed = (int(Random(5)) + 2) *
        ➥(1 - (random(2)* 2));
        bubble[i].yspeed = (int(Random(5)) + 2) *
        ➥(1 - (random(2)* 2));
    }
    originalBubble. _visible = false;
}
```

(The ➥ symbol you see here is for editorial purposes only.)



```
onClipEvent (load) {

    maxBubbles = 4;
    bubble = new Array();

    var screen = _root.scr.getBounds(_root);
    for (var i = 0; i <= maxBubbles; i ++){
        duplicateMovieClip(originalBubble, "bubble" + i, i);
        bubble[i] = eval("bubble" + i);
        bubble[i]._x = random(screen.xMax);
        bubble[i]._y = random(screen.yMax);
        bubble[i].xspeed = (int(Random(5)) + 2) * (1 - (random(2)* 2));
        bubble[i].yspeed = (int(Random(5)) + 2) * (1 - (random(2)* 2));
        bubble[i].isAlive = true;
    }
    originalbubble._visible = false;
}
```

Line 12 of 17, Col 14

The **onClipEvent(load)** code now turns the original Bubble movie clip invisible.

The steps you've taken to initialize the bubble module during the **load** event handler reflect an essential concept in advanced Flash programming. Let's go over it one more time to make sure all the concepts are understood…

You create a loop with a counter variable called ***i*** that counts from zero to a number right below the number stored in ***maxBubbles***.

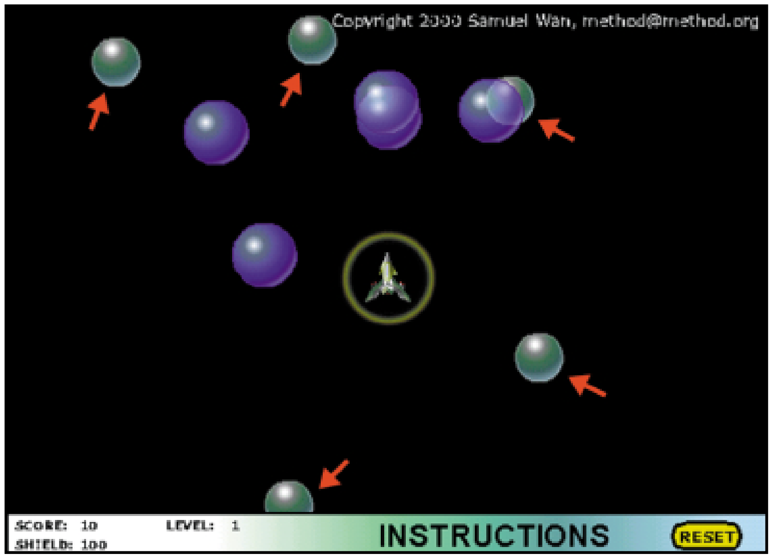
You start with the number zero because the first element in an array is counted as the zero element instead of the first element.

Every time the loop iterates, a new duplicate of the originalBubble Movie Clip is created, and it is dynamically given an instance name of "bubble" with the increasing value of variable ***i*** attached at the end (for example bubble0, bubble1, bubble2, bubble3 and so on). The new duplicate movie is also given a depth of ***i***, because two duplicate movies cannot occupy the same depth in the same timeline.

You assign the new bubble instance to an element of the **bubbles** array according to the increasing ***i*** variable. You then have to use the **eval()** statement to reference an object with the name bubble + ***i*** instead of simply creating a string value. The result of running through the **for** loop and associating objects will produce an array with associated bubble instances. For example, the first four elements of the array will be associated like so:

```
Bubble[0] = bubble0;
Bubble[1] = bubble1;
Bubble[2] = bubble2;
Bubble[3] = bubble3;
```

Note: Even though you started with zero as the first element in the array **(bubble[0])**, it's perfectly acceptable to duplicate a movie into a depth of 0, but not a depth with a negative number.

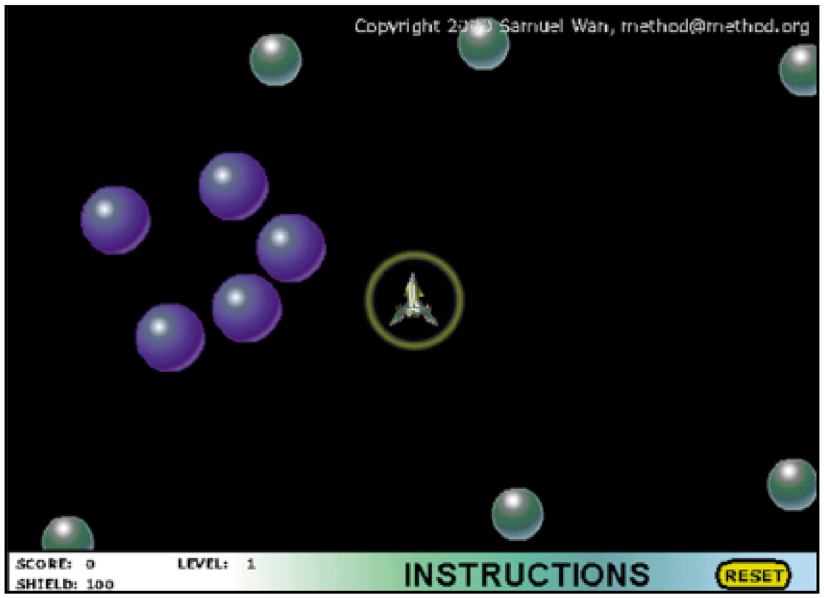


The **onClipEvent(load)** code uses duplicate movie clips to make several copies of the GoodBubble movie clip.

To set the location of the bubbles, you assign a random value between zero and the maximum bounds of the screen movie clip in the main timeline to the *_x* and *_y* values of the bubble instance.

```
bubble[i]._x = random(screen.xMax);
bubble[i]._y = random(screen.yMax);
```

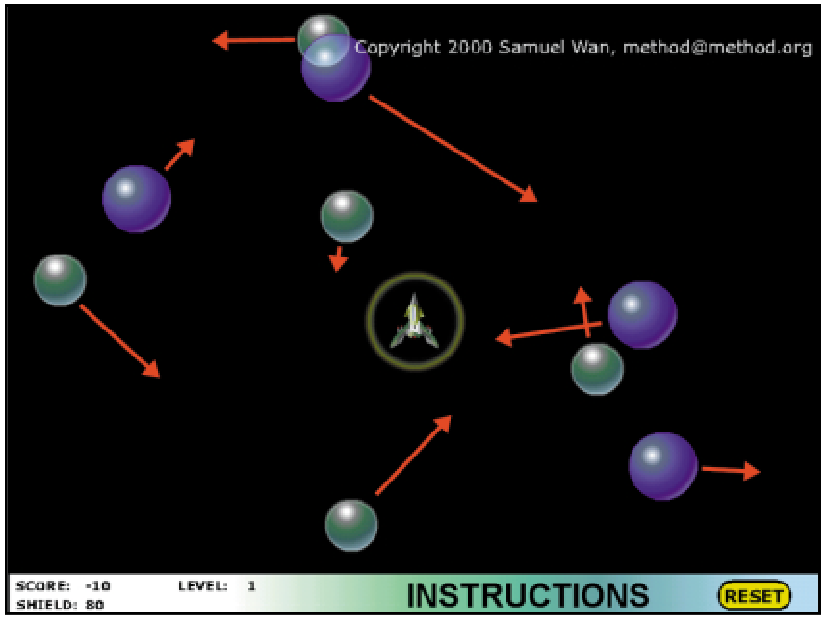
To set the initial speed, you create two variables inside the duplicated movie clip: one for the horizontal speed (***xspeed***) and one for the vertical speed (***yspeed***). Use the formula **(1 - (random(2)* 2)** with the variables to generate a random number between 2 and 6.



Compare this figure to the previous figure (both were shot at the beginning of game play), and you see that the bubbles are randomly positioned at the beginning of the game.

By varying the horizontal and vertical speed of the bubbles, each bubble's direction is likely to be unique. You can add more variety to a bubble's direction by multiplying ***xspeed*** and ***yspeed*** by a positive 1 or –1.

You set the original Bubble to invisible because you need only the new duplicated instances.



Although it's difficult to tell on the printed page, the Bubbles movie offers a variety of different directions and speeds.

Moving the Bubbles

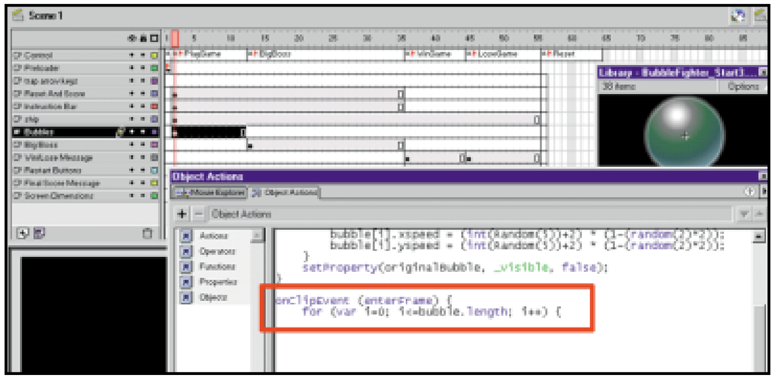
As far as the movement and screen-wrapping goes, the code for the bubble and ship are quite similar. Again, you insert code inside the looping **onClipEvent (enterframe)** event handler to continuously update the position of each bubble and to monitor for any collisions against the ship. The only difference is that the movement and screen-wrapping algorithms refer to elements of an associative array inside a **for** loop instead of a single object. The **for** loop uses the variable ***i*** for a counter again, and the loop iterates until the value of ***i*** reaches the maximum number of elements in the **bubbles** array. We are using the array length as the maximum number of iterations in the loop for a very important reason, which is revealed near the end of this chapter.

1 After the **onClipEvent (load)** handler, add this code to begin the **onClipEvent (enterframe)** handler and the looping:

```
onClipEvent (enterFrame) {
    for (var i=0; i<= bubble.length; i++) {
```

Instantiate the **onClipEvent(enterFrame)** handler and add a **for** loop.

In the previous section, during the **load** event handler, you associated each new duplicated instance of the originalBubble movie clip with an element in the **bubble** array, where bubble0 is associated with **bubble[0]**, bubble1 is associated with **bubble[1]**, bubble2 is associated with **bubble[2]**, and so on. Now you're looping through each array and moving each associated bubble movie clip. Looping from 0 through the length of the **bubble** array will also loop through each element in the **bubble** array.



```
        bubble[i].xspeed = (int(Random(5))+2) * (1-(random(2)*2));
        bubble[i].yspeed = (int(Random(5))+2) * (1-(random(2)*2));
    }
    setProperty(originalBubble, _visible, false);
}
onClipEvent (enterFrame) {
    for (var i=0; i<=bubble.length; i++) {
```

The **onClipEvent(enterFrame)** code begins after the **onClipEvent(load)** code.

As you loop through each bubble, you want to move it in the *x* and *y* direction according to its unique randomized speed that you assigned during the **load** event handler (explained at the beginning of this chapter).

2 Add the bubble's ***xspeed*** and ***yspeed*** to the bubble's current *x* and *y* position.

```
//Propulsion
bubble[i]._x = bubble[i]._x + bubble[i].xspeed;
bubble[i]._y = bubble[i]._y + bubble[i].yspeed;
```

Insert the code that handles movement or propulsion.

3 Next, you need to apply the exact same screen wrap technique that was explained in Chapter 7. The only difference is that instead of screen-wrapping the ship, you'll simply replace all references to the ship with references to the current bubble element, **bubble[i]**.

Remember, when you use the reference to an array element as **bubble[i]**, the value of variable ***i*** increments (increases by 1) during each loop, so all the code within the loop will affect all the duplicated instances of the bubble movie clips.

```
        //Screen wrap
        var screen = _root.screen.getBounds(_root);
        var bubbleBounds = bubble[i].getBounds(_root);
        var localscreen = _root.screen.getBounds(this);
        if (bubbleBounds.yMax < screen.yMin) {
                bubble[i]._y = localscreen.yMax;
        }
        if (bubbleBounds.yMin > screen.yMax) {
                bubble[i]._y = localscreen.yMin;
        }
        if (bubbleBounds.xMax < screen.xMin) {
                bubble[i]._x = localscreen.xMax;
        }
        if (bubbleBounds.xMin > screen.xMax) {
                bubble[i]._x = localscreen.xMin;
        }
}
}
```

Add the code that handles the screen wrapping for the bubbles.

Inserting the ActionScript for the Actual Collision Detection

Now that you have a way to conveniently reference all the bubbles within the module, you can also apply the **hitTest()** method within the **for** loop to detect for collision detection of each of the bubbles against the ship.

1 Apply this ActionScript right after the screen-wrap code inside the **enterframe** event handler:

```
//Collision Detection
if (bubble[i].hitTest(_root.ship.Hull)) {
        bubble[i].gotoAndPlay(2); // pop bubble
}
```

Add the beginning of the collision detection code to the **onClipEvent(enterFrame)** handler.

The code goes into the bottom of the **enterFrame** event handler code so it fits like this:

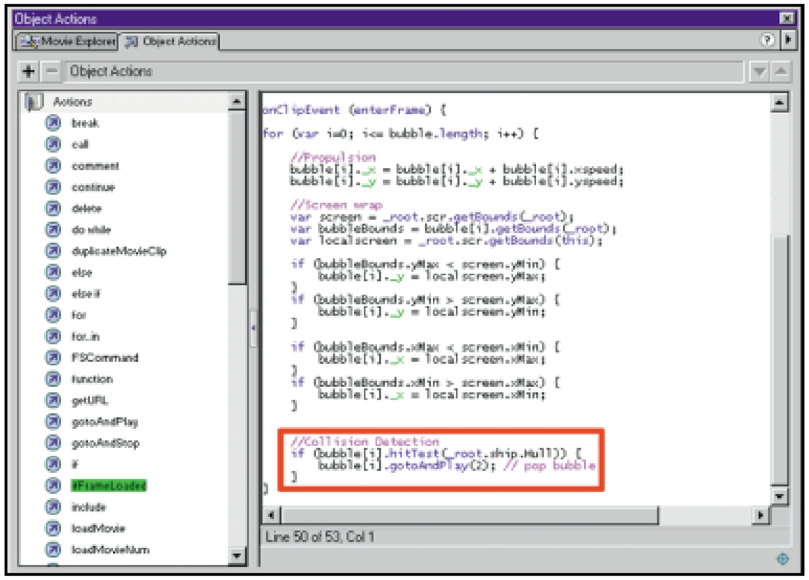
```
onClipEvent (enterFrame) {
for (var i=0; i<= bubble.length; i++) {
    //Propulsion
    bubble[i]._x = bubble[i]._x + bubble[i].xspeed;
    bubble[i]._y = bubble[i]._y + bubble[i].yspeed;
```

```
        //Screen wrap
        var screen = _root.screen.getBounds(_root);
        var bubbleBounds = bubble[i].getBounds(_root);
        var localscreen = _root.screen.getBounds(this);
        •
        •
        •
        if (bubbleBounds.xMin > screen.xMax) {
            bubble[i]._x = localscreen.xMin;
        }
        //Collision Detection
        if (bubble[i].hitTest(_root.ship.Hull)) {
            bubble[i].gotoAndPlay(2); // pop bubble
        }
    }
```

Each iteration of the loop causes each bubble instance to run a **hitTest** against the ship's main structure, the Hull. If a collision is detected, the bubble movie clip goes to a frame with the pop graphics. If you take a peek inside the Good Bubble Unit, which has an instance name of originalBubble, you will see that the second frame, which contains the popping bubble image, begins to play until the third frame. The third frame contains the command **removeMovieClip (this)**, which deletes that instance of the duplicated bubble movie clip. This structure allows you to choose how many frames to display the popped image before removing the bubble itself. Keep in mind that an original movie clip (one that wasn't duplicated from another clip) cannot be removed using the **removeMovieClip** command.

In the code you just inserted, there is an **if** conditional with a **hitTest** function that determines whether the bubble collided with the ship. The line of code inside that **if** conditional tells the bubble movie clip to go to the second frame.

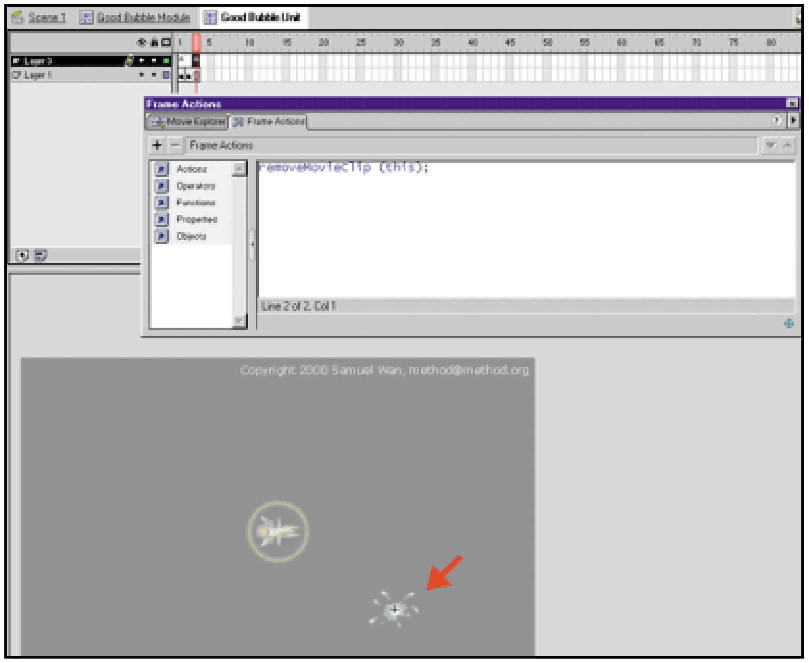
```
  bubble[i].gotoAndPlay(2); // pop bubble
```



```
onClipEvent (enterFrame) {
for (var i=0; i<= bubble.length; i++) {
    //Propulsion
    bubble[i]._x = bubble[i]._x + bubble[i].xspeed;
    bubble[i]._y = bubble[i]._y + bubble[i].yspeed;

    //Screen wrap
    var screen = _root.scr.getBounds(_root);
    var bubbleBounds = bubble[i].getBounds(_root);
    var localscreen = _root.scr.getBounds(this);

    if (bubbleBounds.yMax < screen.yMin) {
        bubble[i]._y = localscreen.yMax;
    }
    if (bubbleBounds.yMin > screen.yMax) {
        bubble[i]._y = localscreen.yMin;
    }

    if (bubbleBounds.xMax < screen.xMin) {
        bubble[i]._x = localscreen.xMax;
    }
    if (bubbleBounds.xMin > screen.xMax) {
        bubble[i]._x = localscreen.xMin;
    }

    //Collision Detection
    if (bubble[i].hitTest(_root.ship.Hull)) {
        bubble[i].gotoAndPlay(2); // pop bubble
    }
}
```

Line 50 of 53, Col 1

The first few lines of the collision detection code are entered.



```
removeMovieClip (this);
```

Line 2 of 2, Col 1

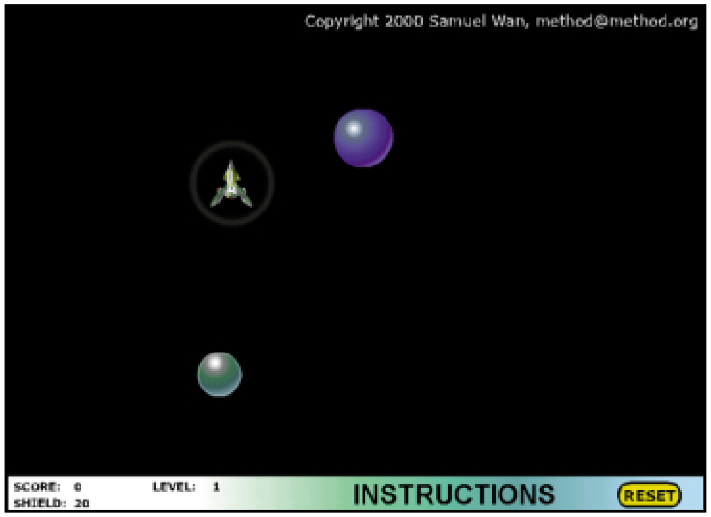
The third frame of the GoodBubbleModel movie clip contains a pop graphic and code that removes or deletes the movie clip.

2 Just below that **gotoAndPlay** command, inside the **if** conditional, add a **splice** method to remove that associated element from the array.

```
bubble.splice(i, 1); //remove associated element from the array
```

Add the **bubble.splice** code to the **Collision Detection** code within the **onClipEvent(enterFrame)** handler.

The **splice** method simply removes an element from an array and decreases the length of the array by the number of elements removed. You remove the element from the array because that bubble is popped, so it saves the CPU power if it doesn't have to check bubbles that have been popped already. Because you've already set the loop to stop as soon as the ***i*** counter reaches the array length, the number of iterations in the loop will decrease as the bubbles are removed from the array. A gradually reduced number of iterations allows the game to run more smoothly because the CPU gradually has to do less work.



Removing bubbles as the ship collides with them saves CPU power because they no longer have to continually be accounted for.

3 Add one more line of code after the **splice** command to add ten points to the score if the good bubble is hit.

```
        _root.score = _root.score + 10;
}
```

Add the **_root.score** code to the **Collision Detection** code.

4 Add some code to check whether the game is over. If the **bubbles** array has reached zero, that means that all the bubbles have been popped, and all the associated elements in the array have been spliced out.

```
        //Is Game Over?
        if (bubble.length == 0) {
           _root.gotoAndStop("BigBoss");
        }
```

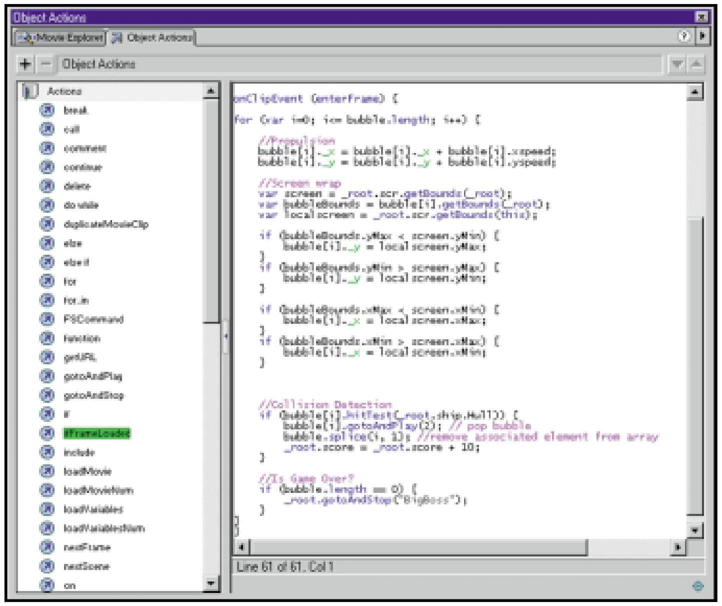
Add this code to the **onClipEvent(enterFrame)** handler below the **Collision Detection** code.

Your complete script for the GoodBubble Module should look like this:

```
onClipEvent(load){
        maxBubbles = 5;
        bubble = new Array( );
        var screen = _root.screen.getBounds(_root);
        ➥onClipEvent (load) {
        for (var i = 0; i < maxBubbles; i ++){
                duplicateMovieClip (originalBubble,
                ➥"bubble" + i, i);
                bubble[i] = eval("bubble" + i);
                bubble[i]._x = random(screen.xMax);
                bubble[i]._y = random(screen.yMax);
                bubble[i].xspeed = (int(Random(5)) + 2) * (1 -
                ➥(random(2)* 2));
                bubble[i].yspeed = (int(Random(5)) + 2) * (1 -
                ➥(random(2)* 2));
        }
        originalBubble. _visible = false;
}
onClipEvent (enterFrame) {
        for (var i=0; i<= bubble.length; i++) {
                 //Propulsion
                 bubble[i]._x = bubble[i]._x + bubble[i].xspeed;
                 bubble[i]._y = bubble[i]._y + bubble[i].yspeed;

                //Screen wrap
                var screen = _root.screen.getBounds(_root);
                var bubbleBounds = bubble[i].getBounds(_root);
                var localscreen = _root.screen.getBounds(this);
                if (bubbleBounds.yMax < screen.yMin) {
                        bubble[i]._y = localscreen.yMax;
                }
                if (bubbleBounds.yMin > screen.yMax) {
                        bubble[i]._y = localscreen.yMin;
                }
                if (bubbleBounds.xMax < screen.xMin) {
                        bubble[i]._x = localscreen.xMax;
                }
                if (bubbleBounds.xMin > screen.xMax) {
                        bubble[i]._x = localscreen.xMin;
                }
                //Collision Detection
                if (bubble[i].hitTest(_root.ship.Hull)) {
                        bubble[i].gotoAndPlay(2); // pop bubble
                        bubble.splice(i, 1);
                        _root.score = _root.score + 10;
                }

                //Is Game Over?
                if (bubble.length == 0) {
                        _root.gotoAndStop("BigBoss");
                }
        }
}
```

(The ➥ symbol you see here is for editorial purposes only.)

Note: The ActionScript for the good bubble module and the bad bubble module look nearly identical except for the code triggered by a collision detection, because hitting either kind of bubble with the ship will have different consequences in the game. This is a good example of how solid modular programming allows the same portions of script to be reused for similar tasks.



```
onClipEvent (enterFrame) {
for (var i=0; i<= bubble.length; i++) {
    //Propulsion
    bubble[i]._x = bubble[i]._x + bubble[i].xspeed;
    bubble[i]._y = bubble[i]._y + bubble[i].yspeed;

    //Screen wrap
    var screen = _root.scr.getBounds(_root);
    var bubbleBounds = bubble[i].getBounds(_root);
    var localscreen = _root.scr.getBounds(this);

    if (bubbleBounds.yMax < screen.yMin) {
        bubble[i]._y = localscreen.yMax;
    }
    if (bubbleBounds.yMin > screen.yMax) {
        bubble[i]._y = localscreen.yMin;
    }

    if (bubbleBounds.xMax < screen.xMin) {
        bubble[i]._x = localscreen.xMax;
    }
    if (bubbleBounds.xMin > screen.xMax) {
        bubble[i]._x = localscreen.xMin;
    }


    //Collision Detection
    if (bubble[i].hitTest(_root.ship.Hull)) {
        bubble[i].gotoAndPlay(2); // pop bubble
        bubble.splice(i, 1); //remove associated element from array
        _root.score = _root.score + 10;
    }

    //Is Game Over?
    if (bubble.length == 0) {
        _root.gotoAndStop("BigBoss");
    }
}
}
```

Line 61 of 61, Col 1

Your completed **onClipEvent(enterFrame)** handler should look like this after you add the **Game Over** code.

Reusing Code

In this section, you quickly and painlessly copy, paste, and modify that same code for the BadBubble module.

1 Select all the code in the Action panel for the Good Bubble Module, and then copy the code into the Clipboard by using Edit > Copy. Close the ActionScript panel.

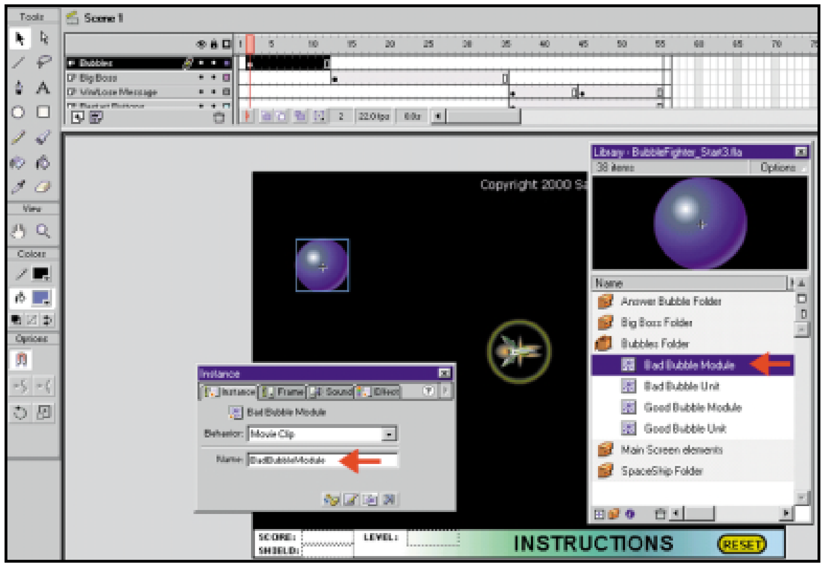
2 Open the library and locate the Bad Bubble Module in the Bubbles Folder. Select the second frame of the Bubbles layer on the same timeline, the same frame that contains the Good Bubbles Module. Drag a copy of this movie clip onto the Stage and give it an instance name of **BadBubbleModule**.

3 Open the Action panel for the BadBubbleModule instance, put the cursor inside the Action panel, and choose Edit > Paste.

Voila, you have just copied and pasted all the code from the Good Bubble Module into the Bad Bubble Module.

4 Look at the **if** statement inside the **//Collision Detection** section. Instead of adding points each time a bubble is popped, you need to cause some damage to the ship. Right after the **splice** method, add code to show that the shield "sparks" a bit, because these are "bad" bubbles that are supposed to damage the ship. Because it damages the ship, you also have to reduce the shield transparency to show that it's weakening the shield. To do so, reduce the shield transparency by 10, and then reduce the hitpoints by 20.

If the bad bubbles (the bigger blue bubbles) hit the ship too many times, the ship should lose its shield, and the player loses the game. If the player loses the game, the Flash movie goes to a frame called LoseGame that informs the player of the bad news. Because the LoseGame frame is text, you should also

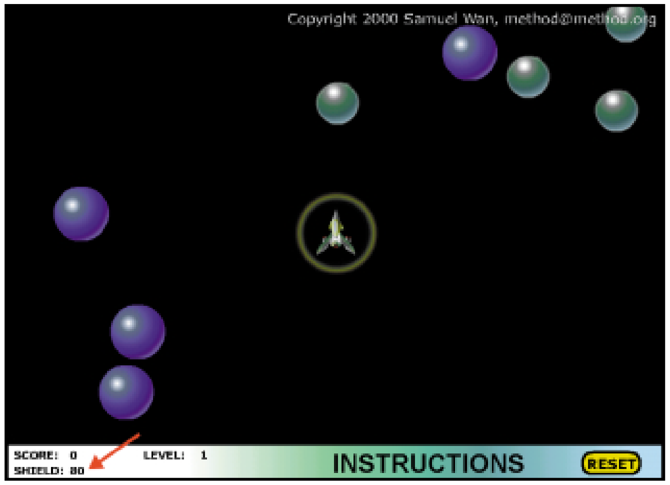


Drag an instance of the Bad Bubble Module movie clip onto the Stage on the Bubbles layer and give it an instance name of **BadBubbleModule**.

```
//Collision Detection
if (bubble[i].hitTest(_root.ship.Hull)) {
        bubble[i].gotoAndPlay(2); // pop bubble
        bubble.splice(i, 1);

        //Reduce shields
        _root.ship.shield.gotoAndPlay(2); //Show shield sparks
        _root.ship.shield._alpha -= 20; //Reduce shield transparency
        _root.score = _root.score - 10; //Reduce score by 10 points.
        _root.ship.hitpoints -= 20;
}

//Is Game Over?
if (bubble.length == 0) {
        _root.gotoAndStop("BigBoss");
}
```

Modify the code that you copied and pasted from the GoodBubbleModule movie clip instance to reduce the shield and reduce points.

toggle the movie to high quality, so the text will show up anti-aliased and will be easy to read. You will make such changes to the **//Is Game Over?** section of the code.

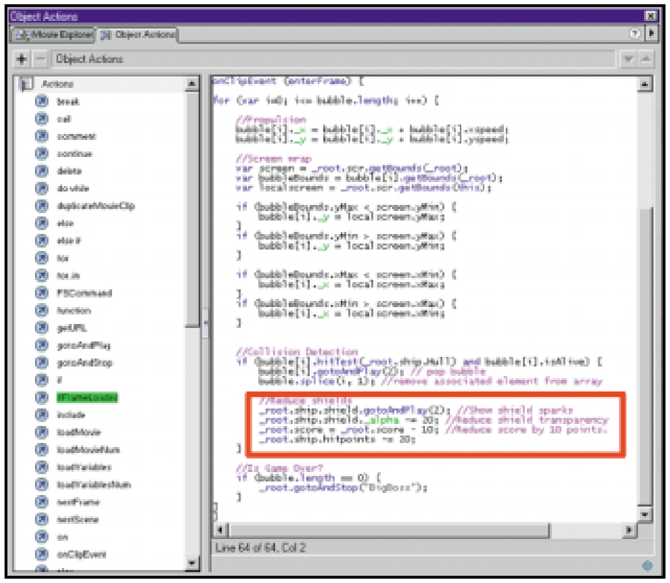


The Shield value is reduced every time the ship collides with a bad bubble.

5 Find the following code from the original Good Bubble Module version:

```
//Is Game Over?
if (_root.ship.hitpoints <= 0) { //Remember to change to
➥reflect shield strength
    _root._highquality = 1;
    _root.gotoAndStop("LoseGame");
}
```

(The ➥ symbol you see here is for editorial purposes only.)



```
onClipEvent (enterFrame) {
for (var i=0; i<=bubble.length; i++) {

    //Propulsion
    bubble[i]._x = bubble[i]._x + bubble[i].xspeed;
    bubble[i]._y = bubble[i]._y + bubble[i].yspeed;

    //Screen wrap
    var screen = _root.scr.getBounds(_root);
    var bubbleBounds = bubble[i].getBounds(_root);
    var localscreen = _root.scr.getBounds(this);

    if (bubbleBounds.yMax < screen.yMin) {
        bubble[i]._y = localscreen.yMax;
    }
    if (bubbleBounds.yMin > screen.yMax) {
        bubble[i]._y = localscreen.yMin;
    }

    if (bubbleBounds.xMax < screen.xMin) {
        bubble[i]._x = localscreen.xMax;
    }
    if (bubbleBounds.xMin > screen.xMax) {
        bubble[i]._x = localscreen.xMin;
    }


    //Collision Detection
    if (bubble[i].hitTest(_root.ship.Hull) and bubble[i].isAlive) {
        bubble[i].gotoAndPlay(2); // pop bubble
        bubble.splice(i, 1); //remove associated element from array

        //Reduce shields
        _root.ship.shield.gotoAndPlay(2); //Show shield sparks
        _root.ship.shield._alpha -= 20; //Reduce shield transparency
        _root.score = _root.score - 10; //Reduce score by 10 points.
        _root.ship.hitpoints -= 20;
    }
    //Is Game Over?
    if (bubble.length == 0) {
        _root.gotoAndStop("BigBoss");
    }
}
}
```

Modify the **Collision Detection** code as shown.

6 Replace the selected code with the Bad Bubble Module version:

```
//Is Game Over?
if (_root.ship.hitpoints <= 0) {
    //Remember to change to reflect shield strength
    _root._highquality = 1;
    _root.gotoAndStop("LoseGame");
}
```

Modify the **Game Over** code to ensure high-quality viewing and to display the LoseGame frame instead of BigBoss.

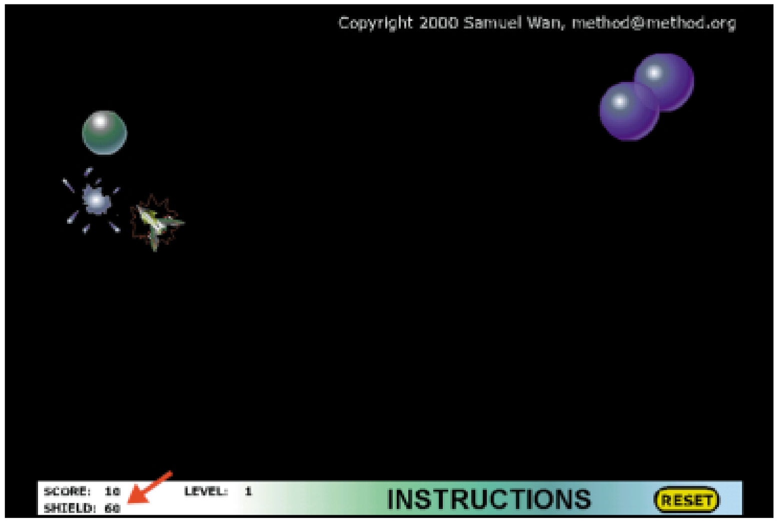
The final code for the BadBubbleModule should look like this:

```
onClipEvent (load) {
     maxBubbles = 5;
     bubble = new Array( );
     var screen = _root.screen.getBounds(_root);
     for (var i = 0; i < maxBubbles; i ++){
          duplicateMovieClip (originalBubble, "bubble" + i, i);
          bubble[i] = eval("bubble" + i);
          bubble[i]._x = random(screen.xMax);
          bubble[i]._y = random(screen.yMax);
          bubble[i].xspeed = (int(Random(5)) + 2) * (1 - (random(2)* 2));
          bubble[i].yspeed = (int(Random(5)) + 2) * (1 - (random(2)* 2));
     }
     originalBubble. _visible = false;
}

onClipEvent (enterFrame) {
     for (var i=0; i<= bubble.length; i++) {
                //Propulsion
                bubble[i]._x = bubble[i]._x + bubble[i].xspeed;
                bubble[i]._y = bubble[i]._y + bubble[i].yspeed;

                //Screen wrap
                var screen = _root.screen.getBounds(_root);
                var bubbleBounds = bubble[i].getBounds(_root);
                var localscreen = _root.screen.getBounds(this);
                if (bubbleBounds.yMax < screen.yMin) {
                     bubble[i]._y = localscreen.yMax;
                }
     if (bubbleBounds.yMin > screen.yMax) {
          bubble[i]._y = localscreen.yMin;
     }
     if (bubbleBounds.xMax < screen.xMin) {
          bubble[i]._x = localscreen.xMax;
     }
     if (bubbleBounds.xMin > screen.xMax) {
          bubble[i]._x = localscreen.xMin;
     }
     //Collision Detection
     if (bubble[i].hitTest(_root.ship.Hull)) {
          bubble[i].gotoAndPlay(2); // pop bubble
          bubble.splice(i, 1);
          //Reduce shields
          _root.ship.shield.gotoAndPlay(2); //Show shield sparks
          _root.ship.shield._alpha -= 20; //Reduce shield transparency
          _root.score = _root.score - 10; //Reduce score by 10 points.
          _root.ship.hitpoints -= 20;
     }

     //Is Game Over?
     if (_root.ship.hitpoints <= 0) { //Remember to change to reflect shield strength
          _root._highquality = 1;
          _root.gotoAndStop("LoseGame");
     }
}
```

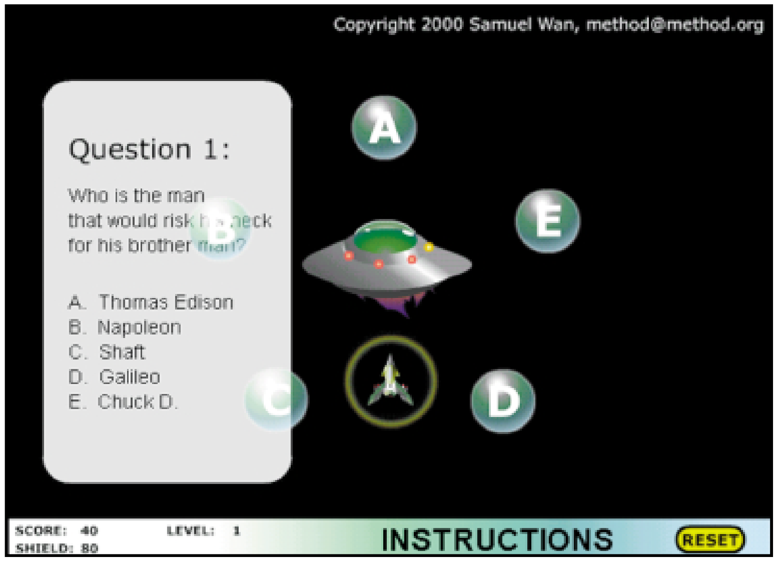
How It Works

If this chapter were simply titled "Collision Detection," it would require only a paragraph explaining the **hitTest(target)** method. What makes this chapter worth reading is the fact that we've written code to detect collisions for multiple projectiles. To program algorithms for multiple objects (such as many bubbles flying around at once), you used a technique called "associative arrays." This type of array contains an element that refers to another object, such as a movie clip, and it is an elegant way to apply the same block of code to many associated movie clips by running through each element of the array inside a loop.

The two steps happened simultaneously to duplicate the originalBubble movie clip and to associate each duplicate instance to an element in the **bubbles** array. These two steps occurred inside the **onClipEvent(load)** event handler so they would execute only once, right after the bubble modules had loaded into the Flash Player. You initialized each movie clip by assigning it a randomized ***xspeed*** and ***yspeed***, and then you set each movie clip to a random location on the screen.



The ship's shield loses strength when you collide with a bad bubble.

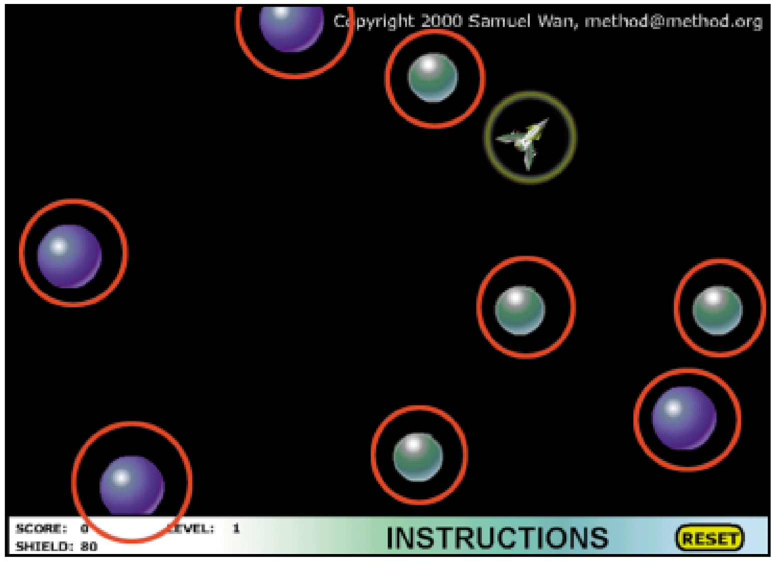


Question 1:

Who is the man that would risk his neck for his brother man?

A. Thomas Edison
B. Napoleon
C. Shaft
D. Galileo
E. Chuck D.

The player gets to go to the Big Boss screen if he removes all the good bubbles (the green ones).



The Collision Detection engine must account for a number of objects or an array of objects.

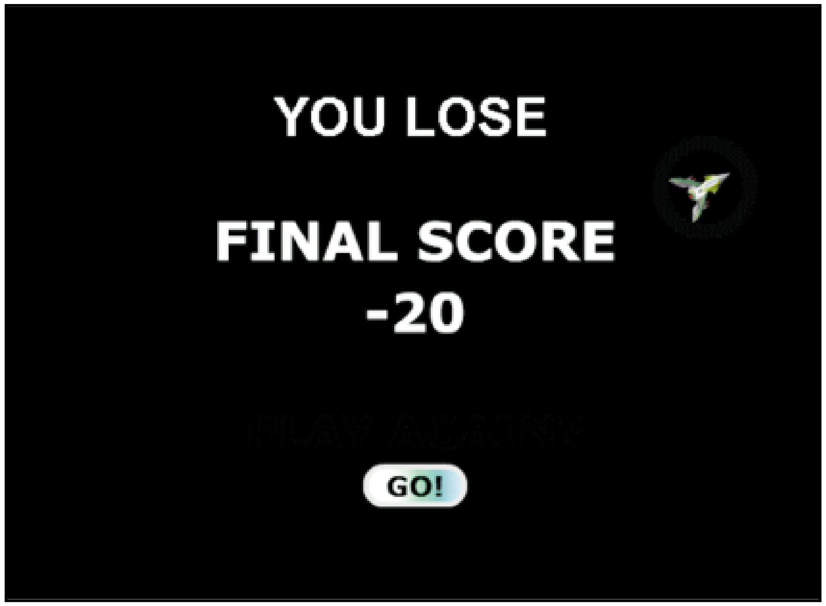
The **onClipEvent(enterFrame)** handler contained all the code executed continually to move the bubbles and detect for collision. You used the same screen-wrap code with slight modification for the bubbles, too.

Several events occur in sequence when a collision is detected between a bubble and the ship's hull:

1. The bubble's associated element in the **bubbles** array is spliced (or removed) from the array.
2. The bubble goes to frame two and begins playing until frame three.
3. The ActionScript in frame three causes the bubble movie clip to remove itself.

If the ship has collided with a "good" bubble, the collision adds 10 points to the score. If it has collided with a "bad" bubble, the collision reduces the shield by 20 points.

Finally, you copied and pasted the complete code from the GoodBubbleModule to the BadBubbleModule and made certain adjustments to the code to reflect the behavior of the two different kinds of bubbles.



The player is sent to the Loser screen if he removes all the bad bubbles (the blue ones).