Written by Billy D. Spelchan for www.BlazingGames.com

Chapter 8

Bomb NIM

Contents

The second game that we are creating in this part of the book is a variation of the first game. The big difference is the way that we do the game. This game uses a lot more Action Script to manage the game.

- Building a Bomb - Deciding on the type of bomb.
- Lighting the Fuse - Creating an animated fuse.
- Exploding Bombs - Creating three types of explosions.
- Bomb Highlighting - Adding code to highlight a bomb.
- Layout - Laying out the playfield.
- Math Behind Motion - Creating code that moves a movie.
- Animating the Layout - An animated layout sequence.
- Player turn - Handling the player's move.
- Bomb Removal - Removing bombs.
- Computer Turn - Making the computer's move.
- Game Over - Ending the game and the title sequence.

Building a Bomb

As the game obviously is based around bombs, the first thing we are going to need is an image of a bomb. There are a huge number of choices we can choose from. We could have some type of plastic explosive with a red lcd screen on it. If you watch a lot of movies that happen to have bombs in them, then you know that there is some rule stating that LCD's on bombs have to be red. I suppose green is also used, though red is by far the more common color for a bomb's LCD.

If you want a more sinister bomb, then you could go with a nuclear bomb. The advantage of this is that nuclear bomb's can look like anything. This means that you could get away with having a simple box with a nuclear symbol on it and a red lcd on it (see above).

Other potential choices would be dynamite, Molotov cocktails, grenades, or even precision bombing type of bombs (the ones you see dropped out of air planes). I personally like the traditional Saturday morning cartoon bomb. You know, the round bombs with a fuse on the top of them. That is the type. In addition to being recognisable, it is also very easy to draw.

Essentially I just took a circle. Used the gradient tool to create a radial gradient that went from light grey to dark grey. This has the effect of creating what looks like a light spot on the bomb. By using the gradient adjustment tool, we can then stretch out the gradient and move the highlight over to one side of the bomb. The top part of the bomb is simply a rectangle with a linear gradient applied to it. The fuse is simply a rectangle with a solid light yellow applied to it. All three objects are then selected with the arrow tool and the combined object is converted into a graphic symbol.

We are not yet done, however. We want to then select the instance of the bomb symbol and create a new symbol with it. This time it will be a movie symbol. The movie is needed as we want the bombs fuse to be lit, so an animated sparking effect will have to be created.

Lighting the Fuse

One of the most versatile features that Flash possesses is the ability to have movie clips within movies. The amount of flexibility that this adds to Flash is incredible. Not only can you have movie clips within a movie, but it is also possible to have those movie clips contain other movie clips. What makes this even more powerful, is the fact that the movie clips will run independently of each other. That means that the main movie can be stopped or in some type of loop and that action will have no effect on what frame is being played in the movie.

This feature can at times be a disadvantage as you may want a bit more control over the movie clip. This is where the power of Action Script comes into play. When a movie clip is used, an instance of the Movie class is created for that movie. This class has a lot of variables for controlling the movie (see the Action Script dictionary for a complete function listing). Not only can you control the position, size, and orientation of the movie, it is also possible to control the frame. If that was not enough, any functions written within that movie can also be called, with the movie being able to call functions within it's parent's class!

By creating a series of functions that go to appropriate sequences, it is quite easy to create an easily modifiable movie that can be controlled by the parent movie. It also allows you to more easily divide asset creation among multiple team members.

To create the bomb animation, we need to edit the bomb movie. This is done by opening the library and double clicking on the bomb entry. This brings up a movie unique to the bomb, down to having it's own time frame. Initially there is only one layer. We will need six layers for the final version of the bomb movie. I labelled the layers as follows: "Sound", "Code", "Explode", "Fuse", "Main", "Highlight". For now, though, we will only need the "Code", "Fuse", and "Main" layers. The main layer is where the bomb goes. Click on the 10th frame and choose add Frame to make sure the bomb is visible for the first ten frames.

On the "Fuse" layer, create blank keyframes on frames 2, 4, 6, and 8. These will be spark effects. Fill out each of these frames by drawing sparks. I just used short lines of pencil width 2 using red, yellow, and orange for the pen color. Figure 1 show the four spark frames as I drew them.

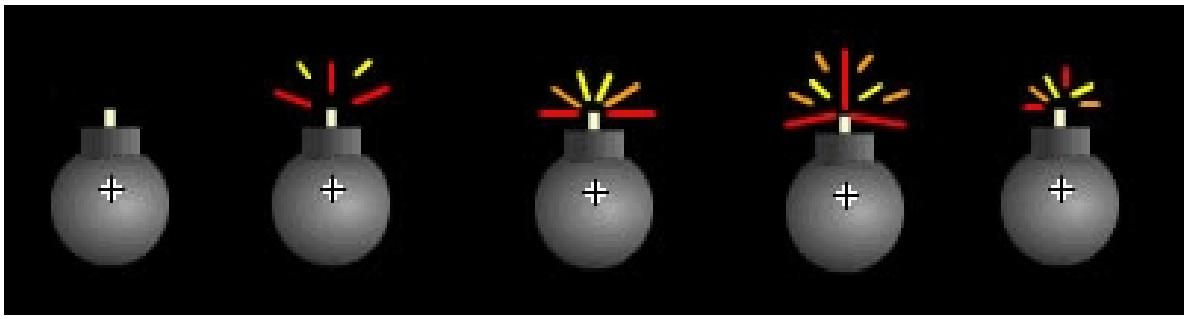

Figure 1: Bomb fuse animation

To make the spark a random effect, some action script is used to randomly change the spark image. This code is on the "Code" layer on frames 3, 5, 7, and 9. Each of these frames has slightly different code to make sure that the next frame is not the frame that is currently playing.

Frame 3's code

```
next = Math.floor(Math.random()*3) * 2 + 4;
gotoAndPlay(next);
```

Frame 5's code

```
next = Math.floor(Math.random()*3) * 2 + 2;
if (next > 2) next += 2;
gotoAndPlay(next);
```

Frame 7's code

```
next = Math.floor(Math.random()*3) * 2 + 2;
if (next > 4) next += 2;
gotoAndPlay(next);
```

Frame 9's code

```
next = Math.floor(Math.random()*3) * 2 + 2;
gotoAndPlay(next);
```

Exploding Bombs

One advantage of using movie clips is the ability to have Action Script within that movie clip. By creating a series of functions within the movie clip, preferably grouped together in one part of the movie clip, you have a class that is extended from the movie class. The functions that are placed in that movie clip should be related to how the movie functions and for control of the movie.

The term API, which stands for Application Programmer Interface, could potentially be used for the series of functions written for a movie clip. Especially if those functions are used to control how the movie clip interacts with the user.

Within our game, our bomb's movie clip has a few important responsibilities. First, it has to have the fuse effect, which we have already completed. Second, it has to support the highlighting of the bomb, which we will be implementing next section. Third, it has to be able to move itself to the appropriate location, which we will also be implementing next section. Finally, and possibly most important, it has to be able to blow up!

Blowing up the bomb is not that difficult, but to make the game more interesting, there will be three different types of explosions. Two functions are going to be created to allow this functionality.

First, we have our reset function. The goal of this function is to return the bomb to a normal state of operations. It simply starts the fuse animation playing again while making the movie visible.

```
function reset()
{
      _visible = true;
      gotoAndPlay(2);
}
```

Now we need a function to actually start the explosion. We want to be able to both randomly select an explosion and go to a specific explosion. As this does not take much additional work, it only makes sense to do this. The function takes a parameter, with the parameter being the type of explosion to show. Using a zero for the explosion type selects an explosion at random. Random explosions are just a random selection between 1 and 3. We then use a switch statement to go to the correct explosion.

```
function explode(how)
{
       var etype = how;
      if (how == 0) etype = Math.floor(Math.random() * 3) + 1;

      switch (etype)
      {
            case 1:
                  gotoAndPlay("flash");
                  break;

            case 2:
                  gotoAndPlay("smoke");
                  break;

            default:
                  gotoAndPlay("cartoon");
                  break;
      }
}
```

The first type of explosion created for the game is a very simple flash explosion. The idea here is to have a bright spot that fades away. This is a very simple explosion as the only thing needed is a big white circle. The circle is converted into an object. The object is placed over the bomb location so it covers the bomb. I cheat here by extending my last bomb frame into the frames for all three explosions and then remove the bomb frames once the explosion animation is done.

First I rapidly expand the white circle. This is done by having a three frame growing motion tween. I then want the flash to turn dark. Alternatively it could be faded out. The darkening effect is simply a tweened hue adjustment. In the code layer, the first frame of the explosion animation should be labelled "flash". The last frame of the explosion animation should be a blank keyframe as there should be nothing at this point. In the code layer, a stop(); statement should be added.

In the sound layer, at the start of the explosion you can add an explosion sound. To add a sound you first need to import a sound file into your library. The sound file can be in any supported format. You then create a blank keyframe on the first frame of the explosion in the sound layer. In the properties panel will be a sound combination box. You simply have to select the soud from that pull down list.

Both the smoke and the cartoon explosions are set up the same way, with the only difference being the image used for the explosion. The images are fairly simple to create. Smoke is just a series of circles. The explosion is simply a bunch of jagged lines. See figure 2 for a look at all three explosion images.

Figure 2: Explosion Types

The sequence here is a five frame growth animation followed by a fading out animation. The growth animation is simply a scaling motion tween. I start with a scaled down version and scale to the 100% version, though starting with a smaller explosion image and scaling it up could also work. The fading is the alpha tweening technique that has been used many times before.

As with the first animation, the first frame should have a label in the code section (using the "smoke" and "cartoon" labels). Likewise, the last frame should be blank with the code layer having a stop() statement. Sound is handled the same way as with the first frame, though you only need to import a sound once.

Bomb Highlighting

The user interface we are planning on using in this version of the game requires that the user select the bombs to remove by actually clicking on the bomb to remove. Only three bombs will be selectable. When the player is over the first bomb selectable, that bomb will be highlighted. When over the second, both the second bomb and the first bomb will be highlighted. When the third bomb is hovered over, all three of the selectable bombs will be highlighted. Handling this effect will require some Action Script, but before we can even think about that we need to have a way of highlighting the bomb.

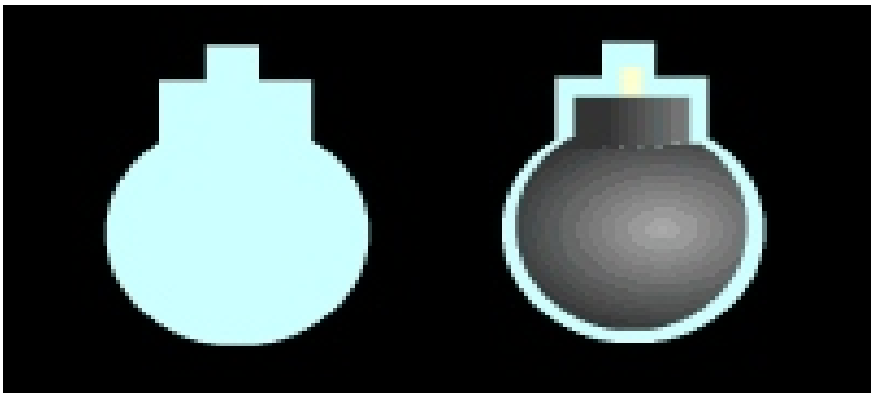

Figure 3: Bomb Highlighting

As you can see by looking at figure 3, the highlight is just a solid colored image that is slightly larger than the bomb but uses the rough shape of the bomb. To create this simply go to the bomb movie. In the highlight layer, draw the rough outline as you see. I used the oval and rectangle tools to do the drawing. Take the complete shape and convert it into a movie symbol. Name the movie symbol highlight_movie. The bomb now has a highlight.

We don't want the highlight visible at all times, so in the code layer of the first frame we add the following line of code:

```
highlight_movie._visible = false;
```

Now we need a way of turning on and off the highlighting. This can be easily solved by adding one function to the bomb movie. This function simply turns on or off the movie's visibility.

```
function setHighlight(state)
{
	highlight_movie._visible = state;
}
```

Layout

We need 40 bombs. This is where the power of Action Script comes into play. Instead of having to create a bunch of separate bombs, by using Action Script's cloning ability, we can easily create a large pile of bombs and place them where we want. One nice thing about the game is that the layout of bombs is very algorithmic.

To start, we go (or create if you don't already have one) to the "Game" scene. Rename the first layer "playfield" and create a second "code" layer. We now need a bomb to clone. Simply drag one onto the side of the playing field and name it “BombBase_movie”.. Before we get to the production code, let us first create a small test to make sure that the layout is the way we want. It is not a bad idea to write test code when developing software.

This test code appears in the code layer of frame 2. Essentially, it creates an array to hold 40 bombs and then loops through each element of the array placing a cloned bomb. The location the bomb is placed is calculated mathematically. The floor function is used to force the value of the modulus or division into an integer. The modulus, or remainder, is used to find the proper column while the divide operation is used to determine which row the bomb belongs to.

Cloning is fairly easy, as there is a function within the movieclip class designed to do just that. What is a bit tricker is adding the clone to an array. This is a bit more complicated than it should be. What we do is create a variable to clone the bomb by adding the counter to a string constant. We then retrieve that variable using eval. the eval function lets you algorithmically build a variable name and then finds that variable for you.

Finally, we move the bomb into it's correct position by changing the clone's _x and _y variables. The final stop statement is there so that we can view our results.

```
var cntr, tempX, tempY;

bombs = new Array(40);
for (cntr = 0; cntr < 40; ++cntr)
{
      tempX = Math.floor(cntr % 8) * 80 + 40;
      tempY = Math.floor(cntr / 8) * 80 + 50;
      BombBase_movie.duplicateMovieClip("bomb"+cntr, cntr+1);
      bombs[cntr] = eval("bomb"+cntr);
      bombs[cntr]._x = tempX;
      bombs[cntr]._y = tempY;
}
stop();
```

Math Behind Motion

One nice aspect of flash is that for most animation, you do not have to worry at all about calculating the positions of moving objects. Flash usually handles the work for you. Sometimes, however, you may want to control the movement of an object yourself using Action Script, like we did with the layout of the bombs.

While you can just use the function I created (it is generic enough that it can be plugged into any Flash movie) it is always nice to know how something works. There are many ways of handling movement, some of which we will discuss in future chapters.

As you have learned in math class, the angle of a line can be expressed as a slope. The slope is simply a fraction consisting of a number indicating how many units the line will move up (the rise) divided by the number of units the line will move to the right (the run). This number can be used for motion as it is essentially represents a ratio of vertical versus horizontal movement. An object with a slope of ½ will move horizontally two units per unit it moves up.

This seems simple enough, and it's use in motion also seems clear, except that there is one problem. What happens if the run of the slope is zero? Another problem, is what if you want the units to be moved to be the total units, not the number of horizontal units? While there are problems with the slope formula that prevents it from being utilized, we can use the concepts behind the slope to build a general purpose function for handling moving an object speed units towards a destination.

The function declaration has three variables. The obj variable would be the object to move. The targetX and targetY variables would be where the object is moving to. Finally, the speed variable would be how far to move towards the destination.

```
function moveObj(obj, targetX, targetY, speed)
{
```

The first thing we need to do is figure how far we are going to be moving horizontally and vertically. To do this we find the delta values of the line. This requires we know the objects starting and ending locations, and then figure out the respective travel lengths along the two axis. Next we need to find the cumulative distance by combining the absolute values of the two delta variables. We use this information to try to keep the number of units travelled correct.

```
      var curX = obj._x;
      var curY = obj._y;
      var deltaX = targetX - curX;
      var deltaY = targetY - curY;
      var distnce = Math.abs(deltaX) + Math.abs(deltaY);
```

Now we check if the distance is less than the amount we wish to travel. If it is, we simply move to the ending location.

```
	if (distance <= speed)
	{
		obj._x = targetX;
		obj._y = targetY;
		return true;
	}
```

If the distance to travel is greater than the speed then we are going to have to calculate how far we travel along each axis by multiplying the delta by the distance to travel divided by the cumulative distance. This may sound confusing so lets try to explain it another way.

We know that we want to move speed units. We want to divide this number of units between the two axis. to do this, we multiply the delta variable, which is the length of the axis, by the ratio of the speed to the total distance of the two axis.

```
	else
	{
		var moveX = deltaX * speed / distance;
		var moveY = deltaY * speed / distance;
		obj._x += moveX;
		obj._y += moveY;
		return false;
	}
}
```

Animating the Layout

At this point we are ready to start writing the layout animation code. The first thing that has to be done is the code we wrote last chapter is going to have to be replaced. We want the bombs to appear along the edges of the screen and move to their final resting spot. In order to do that, we will randomly place the bombs along the border when they are cloned. The second frame's code would then be as follows.

```
var cntr, tempX, tempY, tempR;

bombs = new Array(40);
for (cntr = 0; cntr < 40; ++cntr)
{
      tempR = Math.floor(Math.random()*4);
      switch (tempR)
      {
            case 0:
                  tempX = Math.random() * 640;
                  tempY = 0;
                  break;
            case 1:
                  tempX = 640;
                  tempY = Math.random() * 480;
                  break;
            case 2:
                  tempX = Math.random() * 640;
                  tempY = 480;
                  break;
            default:
                  tempX = 0;
                  tempY = Math.random() * 480;
                  break;
      }
      BombBase_movie.duplicateMovieClip("bomb"+cntr, cntr+1);
      bombs[cntr] = eval("bomb"+cntr);
      bombs[cntr]._x = tempX;
      bombs[cntr]._y = tempY;
}
currentBomb = 0;
```

We are going to have to move the bombs. Here is a generic movement function that we wrote last section slightly changed to better reflect it's use in the game. This simply moves the passed object from it's current position speed units towards it's final position.

```
function moveBomb(bomb, targetX, targetY, speed)
{
      var curX = bomb._x;
      var curY = bomb._y;
      var deltaX = targetX - curX;
      var deltaY = targetY - curY;
      var distance = Math.abs(deltaX) + Math.abs(deltaY);
      if (distance <= speed)
      {
            bomb._x = targetX;
            bomb._y = targetY;
            return true;
      }
      else
      {
            var moveX = deltaX * speed / distance;
            var moveY = deltaY * speed / distance;
            bomb._x += moveX;
            bomb._y += moveY;
            return false;
      }
}
```

Finally we are going to create a label named "layout" on frame five. In frame 6 we have the code for laying out the bombs. This simply loops through all the bombs calling the previously written movement function until all the bombs are in their final position.

```
var tempX, tempY, cntr;
var active = 0;

for (cntr = 0; cntr < 40; ++cntr)
{
      tempX = Math.floor(cntr % 8) * 80 + 40;
      tempY = Math.floor(cntr / 8) * 80 + 50;
      if (moveBomb(bombs[cntr], tempX, tempY, 10) != true)
            ++active;
}
if (active > 0)
      gotoAndPlay("layout");
```

Player turn

While the layout looks really neat, there isn't any type of interface for the game yet. The first thing that has to be done is to create a new labelled section on the "Game" scene. I used frame 10, which I labelled "player". Within this frame I added the following Action Script code.

```
playerPlaying = true;
stop();
```

What this does is sets the variable playerPlaying to true and then stops the movie. The playerPlaying flag will be used to make sure that the player is only able to select bombs when it is his turn. We also want to indicate to the player that it is his or her turn. I created a message layer in the movie and use that layer to have a text message at the bottom of the screen.

To actually handle the selection of bombs, we are going to take advantage of two mouse controlling methods. All this new code is going to be placed in frame 2. First, we append some game initialization code. This code resets some bomb tracking information and makes sure that it is not the players turn. It goes just before the moveBomb function.

```
currentBomb = 0;
playerPlaying = false;
boomCount = 0;
```

Now we have to add the first mouse function. This function is called every time the mouse is moved. If it is not the player's turn, this function simply returns. Otherwise it determines the largest bomb that the player can take. This is determined by adding 2 to the current bomb, but as there are only 40 bombs (numbered from 0 to 39) we have to make sure this number is valid by using the Math.min function. As it's name suggests, this function returns the minimum of the two values passed to it.

We then loop backwards from the last possible bomb the player could take to the first possible bomb the player could take. Using a hitTest function call, we determine if the mouse is over top of the bomb we are checking. If it is, or if a higher bomb had proven to have the mouse over it, we highlight the bomb. Now, a very important thing, if this is not the case, we turn off highlighting! This is required otherwise once a bomb has been highlighted it will stay highlighted even if the mouse is no longer over it.

```
onMouseMove = function()
{
	if (playerPlaying == false)
		return;
	var maxBomb = Math.min(currentBomb + 2, 39);
	var cntr;
	var bombSelected = false;
	for (cntr = maxBomb; cntr >= currentBomb; --cntr)
	{
		if ((bombSelected) || (bombs[cntr].hitTest(_xmouse, _ymouse,
true)))
		{
			bombSelected = true;
			bombs[cntr].setHighlight(true);
		}
		else
		{
			bombs[cntr].setHighlight(false);
		}
	}
}
```

Now that the highlighting is in place, we want to actually remove the bombs that are selected. This is simply done by using the onMouseUp function. Here we again have to make sure this function only runs when it is the player's turn. We then again use the Math.min function to calculate the ending value for a loop. This time we loop forwards, seeing if any of the bombs has been clicked. If a bomb is clicked, we calculate the number of bombs selected and then start the removal movie sequence, which we will be creating next section.

```
onMouseUp = function() {
	if (playerPlaying == false)
		return;
	var maxCount = Math.min(currentBomb + 3, 40);
	var cntr;

	for (cntr = currentBomb; cntr < maxCount; ++cntr)
	{
		if (bombs[cntr].hitTest(_xmouse, _ymouse, true))
		{
			boomCount = cntr - currentBomb + 1;
			playerPlaying == false;
			gotoAndPlay("remove");
		}
	}
}
```

Bomb Removal

Before I do the removal, I am going to get a little bit ahead of myself. When doing removal, I had to decide if I wanted two simple removal sections of the movie, or one combined section. As the simple sections only require a slight bit of code I opted for two sections. Both sections are almost identical so I will cover them both at the same time.

First, we need to add some labels. The "remove" label I placed on frame 20. On frame 30 I created a label called "cturn", which will be used next section. Frame 40 contains the "cremove" label. Labels are used for looping the movie. The remove frame and cremove has the following code:

```
playerPlaying = false;
bombs[currentBomb].explode(0);
--boomCount;
++currentBomb;
```

On the frame just before the remove section ends (frame 29 in my movie) I have the following code, which loops the section until no more bombs are left to explode. It also checks to see if the player has lost the game by taking the last bomb.

```
if (boomCount > 0)
      gotoAndPlay("remove");
if (currentBomb >= 40)
      gotoAndPlay("playerLose");
```

At the end of the cremove section (frame 49 in my movie) there is a similar piece of code, but for the computer.

```
if (boomCount > 0)
      gotoAndPlay("remove");
if (currentBomb >= 40)
      gotoAndPlay("compLose");
else
      gotoAndPlay("player");
```

Computer Turn

The cturn section contains a small bit of code that forms the computer's thought pattern. Simply put, the computer is smart enough not to take the final bomb if it can help it. When not near the end, it takes a random number of bombs.

```
switch (currentBomb)
{
      case 36:
            boomCount = 3;
            break;

      case 37:
            boomCount = 2;
            break;

      case 38:
      case 39: // LOST!
            boomCount = 1;
            break;

      default:
            boomCount = Math.floor(Math.random()*3) + 1;
}
```

We should also place a message at the bottom of the screen letting the player know that the computer is currently playing.

Game Over

Unlike the original Nim game, the player will lose the game if they take the last bomb. This situation was already checked last section, so all that needs to be done is a simple animated sequence. What I did for the animation sequence is use the bomb layer to have the word "Computer" grow from a tiny size to a large size. I then use the message layer to grow the word "WINS!" from a tiny size to a large size. Finally, a continue button is placed on the screen.

The computer loses sequence was built exactly like the player loses sequence, except that instead of "Computer" we use the word "Player".

The continue button is a simple button. This consists of a rounded rectangle with the text "Continue" inside of it. To add a bit of dazzle, we take two bomb objects and add them to either end. This then results in an animated button (yes, they are that easy to make). The rectangle's color will change when the button is down or over.
To enable the button, we use the following code.

```
con_btn.onMouseRelease = function()
{
      gotoAndPlay("Title", "MainTitle");
}
```

This results in the movie going to the title sequence just after the explosion has finished. Now, some of you are asking "what title sequence?" Others are asking "what explosion?" Well, let's create the title screen now so you know exactly what I am talking about.

The title sequence for the game consists of two main parts. First we start off with an introduction animation. This is simply an enlarged bomb which after a few seconds explodes. When the explosion fades, the second part consists of the title text and the start game button.

The bomb exploding sequence is very simple. We simply take a bomb movie and place it in the middle of the screen. This bomb movie is labelled “titlebomb_movie”. We then scale it so it's large. After a few seconds (frame 30), we have a bit of code in the code layer (remember, I always keep the code in a separate layer) call the bomb instance's explode function. It specifies that the cartoon explosion be shown.

```
titlebomb_movie.explode(3);
```

The second part consists of the title image. This is created by drawing a bunch of white circles to form the words "Bomb NIM". I did this to create a more smokey appearance for the title. The next thing is to create the button. This consists of a rounded rectangle with the text "Start the Game" inside of it. To add a bit of dazzle, we take two bomb objects and add them to either end. This then results in an animated button (yes, they are that easy to make). The rectangle's color will change when the button is down or over.

To get the button to work, we have the following code:

```
start_btn.onRelease = function()
{
      gotoAndPlay("Game", 1);
}

stop();
```