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

Chapter 6

NIM

Contents

In this chapter we finally create our first game.

Creation of the Gem

While those of you who are familiar with Flash probably will just skip this section, I am providing this to show how the gem object was created. Figure1 shows the five steps to how the gem was created, but I will describe the process below for the sake of completeness.

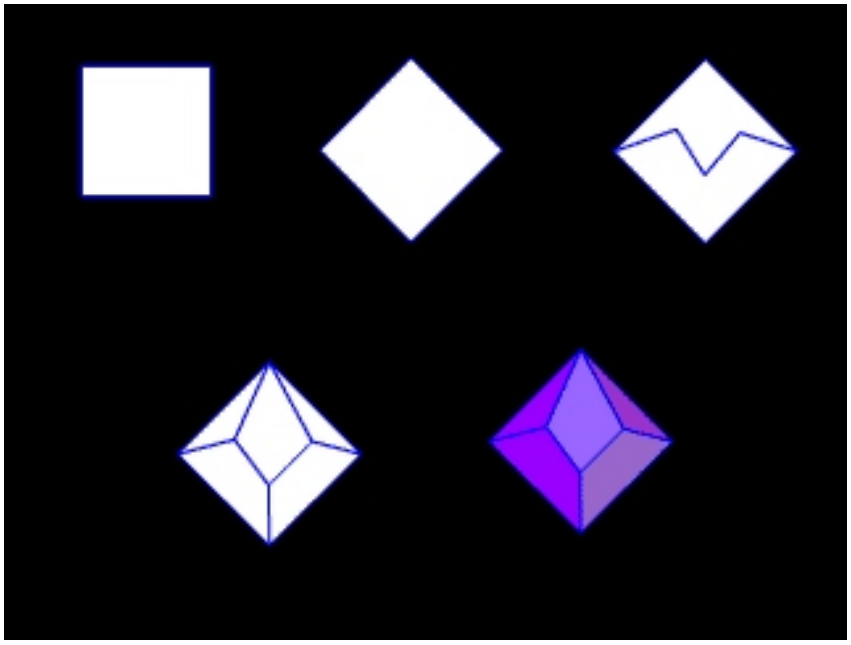

Figure 1: Drawing the Gem

Step 1: Create a square. This is simply a matter of taking the square tool and drawing a square. Make sure the pen color is different from the fill color. Size really isn't important as the image can be scaled to the proper size.

Step 2: Rotate the square 45 degrees. This only requires the selection of the entire square (double clicking, or draw a box around it with the arrow tool) then using the transform panel to rotate the object.

Step 3: Use the line tool to make a M like or W like shape. In the real game I used a W like shape, but for this demonstration, I will go with an M type shape. Simply done by drawing four lines. If you don't like where the lines are, remember that you can adjust the lines by using the subselection (white arrow) tool.

Step 4: Connect the lines to the top and bottom of the gem to form facets.

Step 5: Fill in the individual facets with the colors you like. I like to have lighter colors on one side and darker colors on the other to give a shaded look.

NIM logo

Normally when I create a game, the title and menu screens are the last screens that are created. For this chapter, however, I am going to create the games title sequence first. The title screen is it's own scene. The scene starts off blank. Seven gems then grow onto the screen in what at first seems to be a random pattern. Once the gems are fully grown the letter N fades in behind the gems. This is followed by three gems appearing to create the letter I and another 7 gems to form the M. If you look at the screen shot of the final logo (Figure 2) you will see what the final screen looks like.

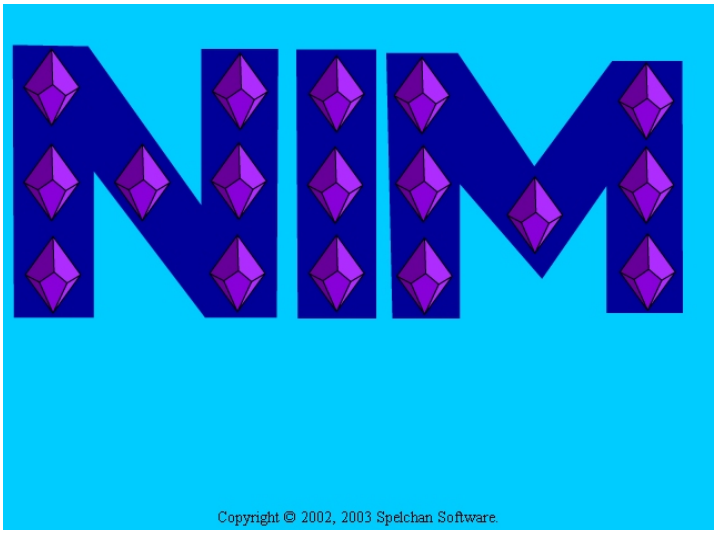

Figure 2 - Logo screen shot

To create this sequence, a large number of layers are used. While the number of layers could be reduced (you will see a technique for that later in this chapter), I figured that for a title sequence, the extra layers wouldn't hurt.

Each gem is on it's own layer. When I created this title, I first layed out all the gems into a rough NIM pattern. To make sure that people could read the title, I then used the line tool to create an outline of the letters. I filled the outlines with a solid color and turned each of the letters into a symbol. At this point I had something similar to Figure 2.

The appearance of the gems are then stepped so that they don't all appear at once, with delays for the appearance of the letters added. A second keyframe is created for each of the gems. The first keyframe is scaled to 10% of the original size. A motion tween between the two keyframe's is then added.

The fading letters is a simple alpha blending tween. You start by creating a second keyframe of the letter. The first keyframe is then set to an alpha level of 0. A motion tween is then placed between the two keyframe's.

Play Game button

The play game button is made by just typing the words "Play the Game" in a large font, converting the font into vector objects by using the break apart command twice and then coverings the whole group of objects into a button symbol. The color of the text is changed for the over and down frames of the button. A solid box that covers the text is used for the hit button. The same technique is used for the “About the Game” and the “Visit Spelchan Software’s website” buttons.

The appearance of the play game button is easily handled by having a keyframe with the button outside the screen and a second keyframe with the button in the final position. A motion tween is added between the two frames. Likewise, the same thing is done with the other two buttons.

For the button to actually do something, a small bit of action script is needed. To associate code with the button, the button has to be given a name. This is done by typing the name into the instance name section of the properties panel (in the left corner of the properties panel). In our case, we are calling the button “play_btn”.

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

This code simply tells the button that it is to execute the command between the curly braces when the button has been released. More technically, the command assigns a function to the button's onRelease variable. As we do not have an existing function, we are creating an inline function by using the function keyword. The command in the braces simply tells Flash to start playing the scene named "Game", which we will be creating in the next section.

The About button is going to be labelled “about_btn” and will play a scene that we will create later in this chapter.

```
about_btn.onRelease = function() { gotoAndPlay("About", 1); };
```

Finally, the visit button, which we will label “spelchan_btn” will open a new web page when it is clicked.

```
spelchan_btn.onRelease = function() { getURL("http://www.Spelchan.com",
"spelchan"); };
```

Limiting Layers

In theory Flash lets you have as many layers as you desire. In reality, however, you should try to limit the number of layers that you use. The technique I used in the title sequence shows you a good way of reducing the number of layers and would work for many situations.There are three main reason that it is a good idea to limit layers: Designer overload, memory, and Flash workload. Lets briefly look at each of these reasons.

Designer overload is caused by the designer having to deal with a huge number of layers. When you have too many layers to deal with, it becomes harder to find the layer you need and if there is interaction between layers that are of substantially different levels, then work can be frustrating. To help prevent this, Flash allows you to place layers into folders. This helps keep isolated elements away from each other and keeps the time line much more manageable.

Every layer takes up memory. Lots of memory is quite common, so this may not be too much of a concern. Still, you never know how old a computer is that uses your program, and with gaming consoles starting to attempt to get into the browser area you never know how little memory a user may have.

Finally, the more layers you have, the more work Flash has to do. Even if the layer is not being animated, Flash still has to check to make sure that the images on that layer are not being obscured by an animated object. While Flash does this dirty rectangle work very efficiently, it still does take time and if a person has a slower computer (or like me has far too many programs running at a time) then that can result in sluggish animation.

To layout the game, we need to get the 40 gems to appear. If you look at Figure 3 you will see what the final layout looks like. The problem here is that 40 gems would result in 40 layers. That is a lot of layers to keep track of. So instead of forty layers, let us group each row into a layer. In fact, we could probably have all forty gems in a single layer, but I originally designed this around four layers.

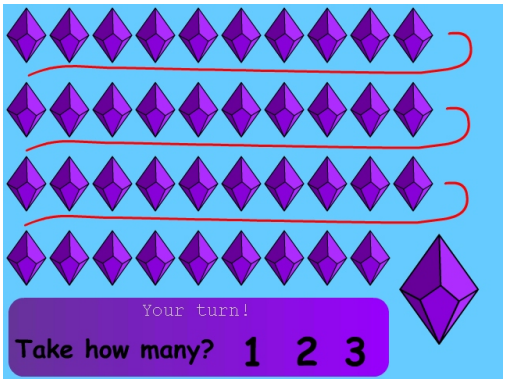

Figure 3: Layout

Five Object Animating Limit

Those of you familiar with Flash probably already know that Flash has a tendency to act funny when you try having animation on a layer that contains more than one object. As that is the case how are we going to animate the gems? After a bit of thought, I realized that at any given time, I would only need five gems animated at a time. This can be handled by creating five animation layers.

The animation is simply handled by assigning gems to animation layers. When the gem reaches it's final position, the layer it belongs to then adds that gem to it's objects. The animation layer is again free to hold another gem.

To demonstrate, lets deal with the first few gems. We create the gems outside the screen in a pattern that wraps clockwise around the screen. The first gem gets assigned to GemAnim1, which is our first animation layer. It uses simple position keyframe animation to move (the first keyframe is where the animation starts, the last keyframe has the gem where the animation ends and a motion tween is placed between the two keyframes). The second gem uses GemAnim2, the third uses GemAnim3 and so fourth. Gem number 6 has no layer, but as it will not start to be animated until gem 1 is in it's final position, it can use GemAnim1. It can do this because there is no longer an object in this animation layer as a gem has been added to the appropriate gem layer. Gem 7 is not needed until gem 2 has been placed in it's layer, so it will use the GemAnim2 layer. And so on, until all forty gems have been animated.

Player Panel

The player panel could have been designed as an all in one object, but that would have added a bit of complexity to the design and I wanted to try and keep this movie as simple as possible for those new to Flash. Instead, the panel is simply a rectangle with rounded corners. It is animated into it's final position by using a simple motion tween.

Once in it's final position we add the components on the panel. The first component on the panel is the message bar at the top of the panel. This is a simple dynamic text object labelled "messages" and set to use a center aligned 24 point grey "_typewriter" font. Below that we create a static text message that reads "Take how many?" I broke this into vector text and created a graphical symbol out of the text. Finally there are three buttons labelled "1," "2," and "3." These three buttons are created the same way we created the "Start the Game" button. Simply type in the number. Break apart the number so you get a Flash object. Select the "Make Symbol" option to convert the object into a button symbol. Fill in the over and down keyframes with different colored versions of the number object. Finally, in the hit keyframe, create a solid box that covers the number.

Once all the components of the panel are visible, we need to make the three buttons actually do something. The code for the buttons is similar to the code that we used with the start game button. The one big difference is that we are using a function that is defined somewhere else. So instead of imbedding the function right into the button's onRelease definition, we are instead telling Flash to use an existing function. Or, at least, we will as soon as we write that code.

```
one_btn.onRelease = take1btn;
two_btn.onRelease = take2btn;
three_btn.onRelease = take3btn;
```

Within the first frame, I have all of my Action Script methods defined. Some programmers recommend using the second frame so the first frame of the animation loads faster, but as this is a separate scene this is irrelevant. The reason the functions are all placed on the same frame is simply to keep all the non-frame specific code in one place. It is a good habit to get into as it makes your movies much easier to debug.

The first function that we will write is a function that sets the message bar text. While this is a very simple thing to do and could be done directly wherever the message needs to be set, I find calling a function that sets the message bar is a bit easier to understand, therefore making the code a bit easier to read. Quite simply, this function takes the text passed to it and assigns that text to the dynamic text object we defined in the panel.

```
function setMessage(s)
{
      messages.text = s;
```

```
}
```

The next function that we are creating is a simple placeholder for a function we will be needing in the future. It simply writes a message to the console saying that we haven't implemented it yet. It will eventually be used to hide the buttons when it is not the players turn.

```
function updateButtons()
{
      trace("updateButtons() not implemented yet");
}
```

Finally we will write the three button handling functions. As you can see, the methods simply set a variable to how many gems are being removed and then calls the updateButtons() function, which will be discussed later. Finally it calls the play() command which causes the movie to start playing. This is important, as the movie is stopped whenever it is the player's turn. Again, this will be covered in more detail later.

```
/*
 * Called when player clicks on the take 1 button
 */
function take1btn()
{
      setMessage("You are taking one gem");
      gemsToRemove = 1;
      updateButtons();
      play();
}

/*
 * Called when player clicks on the take 2 button
 */
function take2btn()
{
      setMessage("You are taking two gems");
      gemsToRemove = 2;
      updateButtons();
      play();
}

/*
 * Called when player clicks on the take 3 button
 */
function take3btn()
{
      setMessage("You are taking three gems");
      gemsToRemove = 3;
      updateButtons();
      play();
}
```

Initializing the Game

At this point, we are now ready to get the game running. To start the game we have a bit of code on the first frame that prepares the game to run. The initialization code simply calls the initialize method and then sets up the games variables. You will notice the variables are not quite correct. This is a bit of a kludge to make the endTurn function easier to write. The end turn function (which will be written later) will remove a gem and swap the players so by setting the player to the computer and having an extra gem, we are actually setting the game up to start with the player and 40 gems. The endTurn function will be discussed later in this chapter

```
initialize();
gemsToRemove = 0;
// :CHEATING CODE:
currentPlayer = PLAYER_COMPUTER; // player will switch imediatly
gemsRemaining = 41; // gem will be removed immediatly
```

The initialize function is designed to run just once. The function then sets up some "constants". These are variables that have been assigned a value that doesn't change. The purpose of these variables is to create more human readable programs. In this case, it provides words for explaining what player is being used.

```
function initialize()
{
      if (PLAYER_PLAYER != undefined)
            return;

      PLAYER_PLAYER = 1;
      PLAYER_COMPUTER = 2;
}
```

Updating buttons

The user interface has one potential problem. The buttons for players actions are accessible at all times. This means that the player can click on buttons while gems are being removed or while the computer is taking it's turn. This can lead to strange game play behaviour. While it is certainly possible to ignore this problem, a little bit of Action Script can solve this problem. This is where the updateButtons function comes in. You have seen this function called when the player clicks on a button. Now we are going to replace this function with one that actually does something. This function sees if it is a proper time for a button to be clicked. If it is, it makes the buttons visible, otherwise it makes the buttons invisible.

```
function updateButtons()
{
	if ( (gemsToRemove > 0) || (currentPlayer == PLAYER_COMPUTER) )
	{
		one_btn._visible = false;
		two_btn._visible = false;
		three_btn._visible = false;
	}
	else
	{
		one_btn._visible = true;
		two_btn._visible = true;
		three_btn._visible = true;
	}
}
```

You control the visibility of a button or movie clip by setting that instance's _visible variable. Setting it to true makes the button visible. Setting it to false not only makes it invisible, but also prevents it from being clicked. Some of you may not be familiar with the term instance. Whenever you have a class, such as a button or movie clip, each newly created object of that class is called an instance. All instances are separate from each other, even though they are from the same class.

The function starts by deciding if the buttons should be hidden. There are two conditions that will cause the buttons to be hidden. First, if gems are in the process of being removed. Second if it is not the players turn. If either of these conditions are met, the buttons are hidden by setting the _visible variable on all three buttons to false. If neither of these conditions are met we assume that the buttons are invisible and make them visible by setting the _visible variable to true.

Ending the Turn

The main endTurn function is designed to handle all of the game logic. This function will be called at the end of each gem removal sequence (which we will be creating in the next section).

```
function endTurn()
{
	--gemsToRemove;
	--gemsRemaining;
	if (gemsToRemove < 1)
	{
		if (currentPlayer == PLAYER_PLAYER)
		{
			currentPlayer = PLAYER_COMPUTER;
			gemsToRemove = Math.floor(Math.random() * 3) + 1;
			if (gemsRemaining < 4)
				gemsToRemove = gemsRemaining;
			setMessage("Computer is taking " + gemsToRemove + " gems");
		}
		else
		{
			currentPlayer = PLAYER_PLAYER;
			setMessage("Your turn!");
		}
	}
	updateButtons();
}
```

The first thing this function does is reduce both the number of gems to remove as well as the gems remaining. It then checks to see if all the gems that were suppose to be removed have in fact been removed. If they have, Flash checks to see what player is currently playing. If it is the human player then the computer player is given it's turn.

The computer takes a random amount of gems unless there are only three or less gems left, in which case it takes the remaining gems. While this sounds simple enough, random numbers are interesting to work with. The Math.random() function returns a number between 0 and 1, with 0 being valid but not ever taking 1. This number is obviously a fraction. We want a whole number between one and three.

The first step in achiving this is multiplying the number by 3. This will result in a number between 0 and just under 3. By using the Math.floor() function, we can round down to the nearest whole number. This results in a 0, 1 or 2. Simply add one to this and we have a number between 1 and 3. As you can see, all this work is done in a single line.

Now we have finished writing all the Action Script functions we need to complete the game.

Gem Removal

The game revolves around the removal of gems, so we obviously need a way of removing gems. As the goal of this movie was to limit the amount of Action Script that is to be used, we will have to animate the removal of all the gems by hand. This is not that difficult of a task, only requiring that we have forty short (10 frame) animation sequences. The question is how do we stop all forty gems from being removed? Unfortunately, this will require a bit of action script at both the start and the end of each removal sequence. At the first frame of the sequence, we need the following code

```
if (gemsToRemove < 1) stop();
```

This code is the same for all forty of the starting frames. It looks at the variable gemsToRemove and sees if the value is less than one, which would mean that there are no more gems to remove. If there is more gems to remove, the movie will simply continue to run, causing the current gem removal sequence to be played. On the last frame of every gem removal sequence, we make a call to the endTurn function we created last section. This adjusts the gemsToRemove variable as well as handles the computer when it is the computer's turn to play.

For the gem removal, I just randomly selected a way of removing gems and manually created that sequence. Before you can remove the gem, you need to remove the gem from it's gem layer and place a gem in the same position as the removed gem but on one of the animation layers. From there, you can freely animate the gem to be removed. I came up with three different ways of removing gems. Shrinking, Fading, and Rockets.

The shrinking technique simply has the gem shrink into nothing. As variations on this technique, some rotation can also be applied to the gem. Essentially you have the first keyframe with the gem at normal size. You then have the last keyframe with the gem at a percentage of what it was, optionally rotated. You then have a Motion Tween between the two frames.

Fading is simply having the motion tween between the gem with 100% alpha and the gem with 0% alpha. Rotation can also be done to this, but with gems near the gem you will want to keep the rotation to only a small amount.

Finally, rocket is simply moving the gem from one location to another off-screen location.

Winning or Losing the Game

As you know, the game ends when the last gem has been taken. Also obvious is the fact that there are two possible outcomes for this game. The player can win the game, or the computer can win the game. Knowing this, our movie is going to have to be able to handle either situation. This will require a bit of Action Script. To be extra precise, a very tiny bit of Action Script, as you will see.

Both sequences start out with the final gem growing larger while moving to the left side of the screen. This is a simple scaling operation combined with a movement operation. The end result is shown in figure 4.
At the end of this sequence, in a labelled sequence called we have the following lines of code.

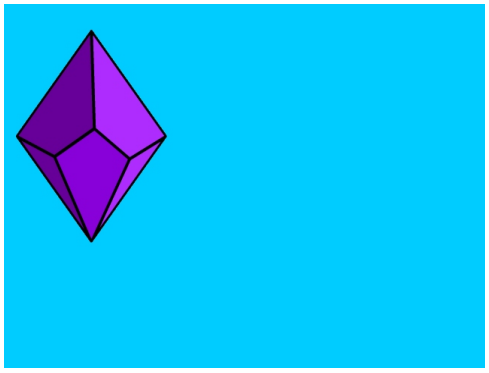

Figure 4: Ending sequence

```
if (currentPlayer == PLAYER_COMPUTER) gotoAndPlay("computerWins");
```

This code is very simple. All it does is see if the current player is the computer. If it is than the computer has won so it will jump forward in the movie to the computer win animation sequence. This is simply a label named "computerWins" which is about thirty frames from the player win sequence. If the player has won, the animation will continue playing, and we happen to be over the playerWins sequence so the player winning message will be shown.

Both sequences simply consist of a text message saying who won the game. Followed by a delay of a couple of seconds. To conclude the sequence, we fade out and then jump to the title sequence by using the following line of code:

```
gotoAndPlay("Title", 1);
```

This variation of the gotoAndPlay function will go to a specified scene (which we can label to be whatever we desire, in this case the label given to the title sequence is Title. The 1 following the scene label indicates that we wish to go to the first frame of that sequence. It is quite possible to go to any frame within this scene.

The About Scene

And now our first game has been completed. All that is left to do is add the about scene. This is simply text that explains the open source licensing. There are two screens tied together by a more button, which is assigned the label "more_btn". The more button is simply a button created the same way the "Play the Game" button was created.

To handle the transition between the first page and the second page we use the following code.

```
more_btn.onRelease = function() { gotoAndPlay("aboutPage2"); };
stop();
```

And to handle the transition between the second page and the title screen we use the following code.

```
more_btn.onRelease = function() { gotoAndPlay("Title", 1); };
stop();
```

As you can see, there is a fair amount of work involved in creating even a simple game.