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

Chapter 30

String Along

Contents

Now we are ready to create our first action game.

- Building the Tile set - the tiles that will be used in this games playfield
- The Basic Playfield - your basic playfield
- Playfield Control - adding the ability to layout the playfield
- Player Layer - making sure the player is present
- Moving Around - using the cursor keys to move
- Collision Detection - bumping into things
- Feeding Frenzy - eating your lunch
- Scoring - handling the games scoring information
- Title - the title page menu

Building the Tile set

We are going to be creating the level layout as a separate movie. While we could simply have the different levels pre-drawn, we are going to travel a more complex route which will prove useful for a lot of more complicated arcade games. What we are going to do is to break the playfield into something known as a tile map. A tile map is simply an array of tiles. Each tile has a property, or set of properties, which controls how the tile acts when the player enters that tile. In this game, there are three types of tiles. There is the empty tile, which has nothing in it and makes up the majority of tiles. There is the obstacle, which the player has to avoid. Finally, there is the energy tile, which the player is trying to pick up.

We could also have the player as part of the playfield, but I have chosen to have the player in a separate layer. Why? Quite simply, I want the player to move smoothly. Having the player as part of the playfield would either greatly complicate the playfield movie or would require that the player only be able to move a whole tile at a time.

Obviously, we are going to need a tile movie that has the three states in it. This is simply a matter of having the tile movie have three labelled frames. The first, empty, would just be a 16x16 square colored dark grey. The second, labelled wall will be a looping animation consisting of a solid white block that has tweened tinting applied to it. This consists of three keyframes, the first and last have no tinting, while the middle keyframe has full tinting. The third label is food and is simply a ball.

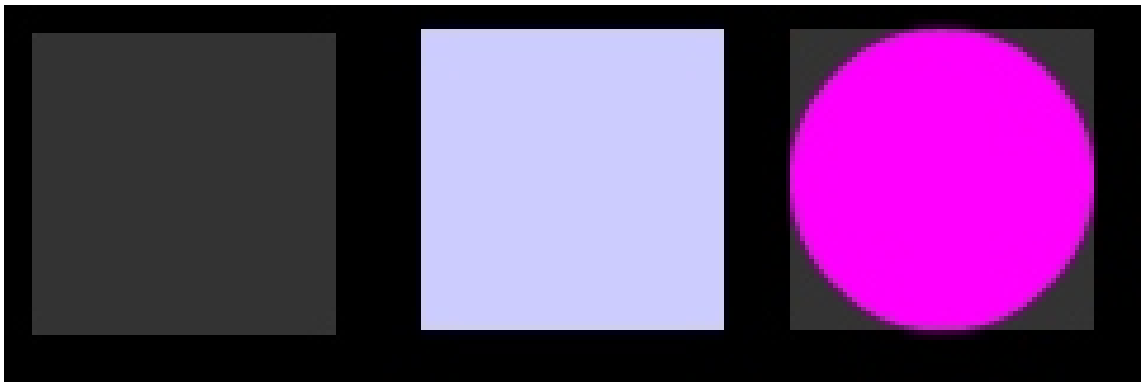

Figure 1: Tiles in the game

For the tile movie to be useful to us, there are two things that we need it to be able to do. The first is to switch the type of tile that it is on demand. I have assigned numeric values to each of the three possible tile states. Constants could have also been set up, but with only three states, I figured that they would be easy enough to remember. The states are as follows:

0 is the Empty state
1 is the Solid or Wall state
2 is the Food state

A function to set the state is simple enough, as all that is needed is for it to store the state value in a variable and set the movie's play head to the appropriate frame. The second requirement is that we need to know what the state of a tile is. The getState function can easily handle this requirement by returning the value of the variable that was set in the setState function.

```
function setState(n)
{
	tileState = n;
	switch (n)
	{
		case 0:
			gotoAndPlay("Empty");
			break;
		case 1:
			gotoAndPlay("Wall");
			break;
		case 2:
			gotoAndPlay("Food");
			break;
	}
}

function getState()
{
	return tileState;
}
```

The Basic Playfield

Now we are ready to assemble the playfield. The basic playfield will be 40x26. This results in an actual playfield size of 640x416. It is useful to create the playfield movie by simply drawing a hollow rectangle that is 640x416 and then turning that hollow rectangle into a movie clip. Just outside the boundaries of the movie clip we will manually place the first tile, which we will name "tile_movie".

The playfield will start by creating the array by cloning the tile we have. To be absolutely sure that the reference tile will not be on the screen, we will make it invisible once the playfield has been created. An alternative way of creating tiles is to change the tile movie's linkage so the tile is exported for Action Script. This would allow you to create the tile using the tile's name. Action Script 2 (which is part of Flash MX 2004) allows you to create an external class file for the movie. While earlier versions of flash theoretically allow for this, it is simply not as convenient and doesn't seem to work as well as doing this in MX 2004 does.

```
initPlayfield();

function initPlayfield()
{
      if (playfieldInitialized != undefined)
            return;
      playfieldInitialized = true;
      PF_ROWS = 26;
      PF_COLS = 40;
      PF_TILESIZE = 16;
      var left = -320 + 8;
      var top = -200;
      pfRows = new Array(PF_ROWS);
      var name, row, temp, tileD = 0;
      for (var cntrY = 0; cntrY < PF_ROWS; ++cntrY)
      {
            pfRows[cntrY] = new Array(PF_COLS);
            for (var cntrX = 0; cntrX < PF_COLS; ++cntrX)
            {
                  ++tileD;
                  name = "t"+ tileD
                  tile_movie.duplicateMovieClip(name, tileD);
                  temp = eval(name);
                  temp._x = cntrX * PF_TILESIZE + left;
                  temp._y = cntrY * PF_TILESIZE + top;
                  pfRows[cntrY][cntrX] = temp;
            }
      }
}
```

Playfield Control

Having a playfield that can vary certainly adds to the game, but we need a way of telling Flash what the playfield looks like. The easiest way would be to send an array containing values for what we want the tiles to be. Creating the arrays would be a bit painful unless we wrote a construction set for building levels. This is an option but is a lot of work for such a simple game.

Another way would be to create a simple level description language and send commands to the playfield which describes what the level looks like. This isn't a bad idea but when you think about it, all we have to do is draw blocks of obstacle tiles. What if we sent the playfield an array that contains a list of rectangles. If the array was null, then the playfield would create an empty playfield, otherwise it would loop through the elements using groups of four elements as the bounds of blocks to draw.

Before creating the code to set up the playfield, we first need a way of resetting all the playfield tiles. As setting all the tiles to empty is equivalent to clearing the playfield, we will write a simple clearPlayfield function. This function simply loops through the rows and columns of the playfield setting the tile in that location to empty unless that tile happens to be along an outer edge in which case a solid wall is created.

```
function clearPlayfield()
{
      for (var cntrY = 0; cntrY < PF_ROWS; ++cntrY)
      {
            for (var cntrX = 0; cntrX < PF_COLS; ++cntrX)
            {
                  if ((cntrX == 0) || (cntrX == PF_COLS-1) || (cntrY == 0) ||
(cntrY == PF_ROWS-1))
                        pfRows[cntrY][cntrX].setTile(1);
                  else
                        pfRows[cntrY][cntrX].setTile(0);
            }
      }
}
```

The code to actually handle the creation of the playfield from a provided array is also simple. The array is simply an array of numbers, with each group of four numbers representing a rectangle using the format x1, y1, x2, y2. For each group of four numbers, we will draw a box by calling the soon to be created drawBox function.

```
function setPlayfield(level)
{
	trace("Setting level...");
	clearPlayfield();
	if (level == null)
		return;

	for (var cntr = 0; cntr < level.length; cntr +=4)
	{
		drawBox(level[cntr], level[cntr+1], level[cntr+2], level[cntr+3]);
	}
}
```

Drawing a box is very easy. We just loop from the starting Y location (y1) to the ending Y location (y2). For each iteration of that loop, we loop from the starting X location (x1) to the ending X location (x2) setting the tile at the x, y location indicated by the loops to solid.

```
function drawBox(x1, y1, x2, y2)
{
	for (var cntrY = y1; cntrY <= y2; ++cntrY)
	{
		for (var cntrX = x1; cntrX <= x2; ++cntrX)
		{
			pfRows[cntrY][cntrX].setTile(1);
		}
	}
}
```

Now we need some code to define the levels. The easiest way of designing levels is to draw out the shape you want the level to be and to go from there. We will go over the design of these levels and the alternative level sets in the next chapter. For now, here is the code for actually using the levels

```
init_game();

function init_game()
{
      if (gameInitialized != undefined)
            return;
      gameInitialized = true;
      num_playfields = 5;
      levels = new Array(
            null,
            new Array(8,13,31,13),
            new Array(8,7,8,19, 8,13,31,13, 31,7,31,19),
            new Array(6,13,33,13, 19,5,19,20),
            new Array(6,13,33,13, 19,5,19,20,
                        9,3,9,10, 3,6,16,6,
                        29,3,29,10, 22,6,36,6,
                        9,16,9,22, 3,19,16,19,
                        29,16,29,22, 22,19,36,19 )

      )

      current_level = 3
}
```

And on frame 2 of the movie, which we will give the label "StartLevel", we add the following code to force the level to be shown and then stop the movie.

```
playfield_movie.setPlayfield(levels[Math.floor(current_level %
num_playfields)]);
stop();
```

Player Layer

The player layer is also a movie clip. For reference, I draw a box the size of the playfield and use that to create the player movie. The main part of the player movie is the string of balls that represent the player. We will use a linked list to handle the string. While we could just use an array (as Flash's arrays have the ability to grow and shrink), we learned about linked lists last chapter so now's our chance to play around with them. The Player movie is simply a 16x16 circle called "string node".

The first thing we have to do in coding the string node movie is to initialize the movie. We will need variables for where the ball was last located as well as where the ball is moving too. We are also going to have a link to the next ball in the chain. While arrays would work, having such a link will make a lot of the functions we will write a bit easier to write (at least someone who is use to working with lists will find the code easier to write).

```
init_stringnode();

function init_stringnode()
{
      if (stringnodeInitialized != undefined)
            return;
      stringnodeInitialized = true;
      last_x = 10;
      last_y = 10;
      target_y = 10;
      target_x = 10;
      nextBall = null;
}
```

We know that the location of the player can change, so we are going to need a function that allows us to move the string node to any location we desire. This quite simply changes the location information.

```
function setLocation(x, y)
{
      _x = x;
      last_x = x;
      target_x = x;
      _y = y;
      last_y = y;
      target_y = y;
}
```

As we are using a simple linked list to handle the nodes of the string, we need a way of clearing that list. This function traverses the list removing all the elements on the list except for the starting element.

```
function clearNodes()
{
      trace ("clearing nodes");
      var lst, nxt;
      nxt = nextBall;
      while ( (nxt != undefined) && (nxt != null) )
      {
            lst = nxt;
            nxt = last.nextBall;
            lst.removeMovieClip();
      }
      nextBall = null;
}
```

Finally, for movement, we are going to need to know where we are heading too. As we already have variables to hold this information, all the function needs to do is set these variables. This is where the power of linked lists and recursion come together. As the string is moving along as a whole, the target for the next node on the list will be the location of the previous node. Knowing this, we just call the next nodes (if there is a next node) setTarget function providing our last location for it's location. The neat thing is, we have just created a recursive function. The next node will call it's next node and so on until there are no more nodes in the string.

```
function setTarget(x, y, speed)
{
      target_x = x;
      target_y = y;
      target_speed = speed;
      last_x = _x;
      last_y = _y;
      if ( (nextBall != undefined) && (nextBall != null) )
            nextBall.setTarget(_x, _y, speed);
}
```

Moving Around

While we have created some of the movement code that will be needed, before any movement can happen two things are needed. First, we need to be able to control the animation on a per-frame basis. This can be done by overriding the onEnterFrame function in the player movie and providing our own playerTick() function. The other function that is needed is the ability to handle the keyboard. Flash has an onKeyDown function that handles key presses. The information about what is happening with the keyboard is stored in the global Key class. Likewise, the Key class has a bunch of constants defined for keys such as the cursor keys. Here is the code for handling the frame rate and for changing direction based on the key pressed.

```
sl = 1;
function onKeyDown()
{
      trace ("keydown");
      if (Key.getCode() == Key.UP)
            setDirection(0);
      else if (Key.getCode() == Key.DOWN)
            setDirection(2);
      else if (Key.getCode() == Key.RIGHT)
            setDirection(1);
      else if (Key.getCode() == Key.LEFT)
            setDirection(3);
      trace("new direction is " + direction);
}

function playerTick()
{
      var done_moving = head_movie.doMove();

      if (done_moving)
      {
            var newx = head_movie._x;
            var newy = head_movie._y;
            switch (direction)
            {
                  case 0: // north
                        newy -= 16;
                        break;
                  case 1: //east
                        newx += 16;
                        break;
                  case 2: // south
                        newy += 16;
                        break;
                  case 3: // west
                        newx -= 16;
                        break;
            }
            head_movie.setTarget(newx, newy, _parent.speed);
      }
}
```

```
function setDirection(n)
{
      direction = n;
}

function startLevel(x, y)
{
      trace ("StartLevel called");
      head_movie.clearNodes();
      head_movie.setLocation(x, y);
      direction = -1;
      onEnterFrame = playerTick;
}
```

As you can see by looking at the code, the animation of the string is done by a method within the string node symbol called doMove. As no such code yet exists we are going to need to write it. The code is very simple. It checks to see if we have reached our target point. If not, it moves the node making sure that the movement will not go past the target. The function then returns true if the movement placed the player on the target location.

```
function doMove()
{
      var newx = _x;
      var newy = _y;

      if (target_x < _x)
      {
            newx = Math.max(_x - target_speed, target_x);
      }
      else if (target_x > _x)
      {
            newx = Math.min(_x + target_speed, target_x);
      }

      if (target_y < _y)
      {
            newy = Math.max(_y - target_speed, target_y);
      }
      else if (target_y > _y)
      {
            newy = Math.min(_y + target_speed, target_y);
      }

      _x = newx;
      _y = newy;
      if ( (nextBall != undefined) && (nextBall != null) )
            nextBall.doMove();

      return ((target_x == _x) && (target_y == _y));
}
```

Collision Detection

There are three types of collisions that we have to worry about. One is with yourself. Two are with the playfield. The first of the playfield collisions is with a wall. The second is with food.

Collision with oneself is easy to check, as you only have to worry about the head coliding with something. Our idea of a collision is when the player attempts to go through a part of ones body. This requires the head go through another location. A simple method can do the check. What this function does is figure out the tile location the target location is. This value is then compared to the last location of each head within the linked list. If one of these locations is the same then a collision has occurred.

```
function checkCollide()
{
      var hit = false;
      var testX, testY;
      var lst, nxt;
      nxt = nextBall;
      var headX = Math.floor(target_x / 16);
      var headY = Math.floor(target_y / 16);
      while ( (nxt != undefined) && (nxt != null) )
      {

            testX = Math.floor(nxt.last_x / 16);
            testY = Math.floor(nxt.last_y / 16);
            if ( (testX == headX) && (testY == headY) )
                  hit = true;
            lst = nxt;
            nxt = lst.nextBall;
      }
      return hit;
}
```

The same principle can be applied to the playfield, though we will need a more complex return value. -1 is a wall, 0 is empty, 1 is food. This function is placed in the playfield symbol. The function simply determines which tile the player is checking. From that, it looks at the type of tile that location contains and returns the appropriate value based on it. As this function will be used only when a tile is entered, we can also use this opportunity to remove the food.

```
function checkTile(x, y)
{
	var tileX = Math.floor((x + 320) / 16);
	var tileY = Math.floor((y + 208) / 16);
	var value = pfRows[tileY][tileX].getTile();
	if (value == 1)
		return -1;
	else if (value == 2)
	{
		pfRows[tileY][tileX].setTile(0);
		return 1;
	}
	else
		return 0;
}
```

All of this collision code is going to need to be called. The best place for this check would be in the playerTick function within the Player symbol. The following code can be added to the end of this function. Quite simply, the code will see if a collision with yourself, a wall, or food has occurred. Hitting yourself or the wall will call a loseLife function in the main time line, which we will be creating shortly. Hitting food will call some functions to cause a new string node to appear and to remove the food from the playfield. These tasks will be done next chapter, but the function calls can be placed here anyway (they just won't do anything as the functions don't exist). I know that having calls to non-existent functions is not a good thing, but in reality, I wrote both these sections together as part of the same programming session, so this complaint really is a false complaint.

```
		if (head_movie.checkCollide())
			_root.loseLife("Hit Self");
		var tile = _root.playfield_movie.checkTile(newx, newy);
		if (tile == -1)
			_root.loseLife("Hit Wall");
		else if (tile == 1)
		{
			addNode();
			_root.eatFood();
		}
```

The last bit of collision detection work that needs to be handled is the loss of a life. The loseLife function is in the main time line and simply stops the level, removes a player, and then moves the play head to the "LoseLife" section, which we will create in a moment.

```
function loseLife(s)
{
      trace(s);
      player_movie.stopLevel();
      --current_lives;
      gotoAndPlay("LoseLife");
}
```

The loose life section contains a simple message that says "Oops!" in large green letters. We skip over 30 frames (one second) and then add a keyframe that contains the following code:

```
updateLabels();
if (current_lives >= 0)
      gotoAndPlay("StartLevel");
```

The updateLabels function will update the score and other player information, but for now will be a placeholder as seen below. The rest of this scripting simply sees if the game is over. If not, it moves the play head to the "StartLevel" section, otherwise the playhead will continue playing.

```
function updateLabels()
{
      // :TODO:
}
```

On the frame following the game over check we have a game over message written in green text. This skips 30 frames where we have another keyframe containing a bit of code that returns the player to the title screen.

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

Feeding Frenzy

In order for the player to eat objects, they need to be placed on the playfield. The way we will do this is with a simple playfield function which we will call addFood. This will work by randomly selecting a location on the playfield. If the location on the playfield is empty, the location will have it's tile type changed to food otherwise another location will be selected.

```
function addFood()
{
	var foodX, foodY;
	do
	{
		foodX = Math.floor(Math.random()*PF_COLS);
		foodY = Math.floor(Math.random()*PF_ROWS);
	}
	while (pfRows[foodY][foodX].getTile() != 0);

	pfRows[foodY][foodX].setTile(2);
}
```

Now it is time to handle the adding of a new string node. This is set up across two frames, to make sure that the newly cloned objects initiate properly (though if we were using a Flash MX 2004 movie, we could get by this problem by having a proper class tied to the string node symbol). To get to the frames we create an add Node function

```
function addNode()
{
	spawining = true;
	gotoAndPlay("spawn");
}
```

On the frame labelled “spawn” we have the following code. Which finds a unique layer and id for the string node and then creates a new string node which will be named tempBall.

```
spawning = true;
var nodeCount = 1000;
var nxt, temp;
 lastNode = head_movie;
next = head_movie.nextBall;
while ( (nxt != undefined) && (nxt != null) )
{
      lastNode = nxt;
      nxt = lastNode.nextBall;
      ++nodeCount;
}
 childname = "node_"+nodeCount;
duplicateMovieClip(head_movie, childname, nodeCount);
tempBall = eval(childname);
```

on the following frame we add the node to the list by calling a new function in the string node called setNext.

```
tempBall.setLocation(lastNode.last_x, lastNode.last_y);
lastNode.setNext(tempBall);
trace("Finished spawing ball " + childname);
gotoAndPlay("PlayerMain");
```

The new functions in the string node symbol are straight forward enough.

```
function setNext(b)
{
      nextBall = b;
}

function getNext()
{
      return nextBall;
}
```

And finally, we need to handle the eating, which, for now is simply a call to the addFood function.

```
function eatFood()
{
      playfield_movie.addFood();
}
```

Scoring

Now we are ready to add scoring to the game. At the beginning of the main movie we add the following code that quite simply sets all the variables to the appropriate values.

```
current_score = 0;
current_lives = 2;
next_life = 10000;
current_remaining = 10;
current_level = 0;
updateLabels();
```

In addition, in order to keep a high score, we add a high score variable which we will define in the games initialization so that it only gets reset the first time the game is played.

```
high_score = 0;
```

We then add a label layer and five labels. Within this layer we create five dynamic text strings. The first, which is in the bottom right of the display, is named “score_txt” and will contain the score of the current game. The second one, named “lives_txt”, contains the number of lives that the player has remaining. The third, on the bottom left, named “high_txt” contains the highest score reached. The “remaining_txt” dynamic text string is located on the top right and will contain the number of pieces of food that the player needs to collect in order to win the level. Finally, the “level_txt” dynamic text string is located in the top center of the screen and contains the current level of the game. The function for updating the labels simply puts proper text into the labels.

```
function updateLabels()
{
      score_txt.text = "SCORE " + current_score;
      lives_txt.text = "LIVES " + current_lives;
      high_txt.text =   "HIGH " + high_score;
      remaining_txt.text = "REMAINING " + current_remaining;
      level_txt.text = "LEVEL " + (current_level+1);
}
```

Points are added when the player eats something. The eatFood function handles this and is replaced as follows.

```
function eatFood()
{
      current_score += (10 * speed_setting);
      if (current_score > next_life)
      {
            next_life += 10000;
            ++current_lives;
      }
      --current_remaining;
      if (current_remaining <= 0)
      {
            player_movie.stopLevel();
            gotoAndPlay("LevelDone");
      }
      else
            playfield_movie.addFood();
      updateLabels();
}
```

Before we add the LevelDone section, lets properly support dying. We replace the script at the end of the loselife section with the following code.

```
if (current_score > high_score)
      high_score = current_score;
gotoAndPlay("Title", 1);
```

We have the level complete section, labelled "LevelDone". This simply has the words “well done!” in large green text. The message lasts for 30 frames and then has the following code.

```
++current_level;
current_remaining = 10 * (Math.floor(current_level / num_playfields)) + 10;
gotoAndPlay("StartLevel");
```

Title

Now we want to do a title sequence. When you think about it, The game is called String Along, so wouldn't it be neat if the title appeared as a string of letters following the leader until they reached their desired locations.

That being said, Let us assemble the title sequence. First we write out the title. I chose to do the word “String” in bold and italic, while the “along” was done in just italic. I then break the text apart twice and then separate into layers. This puts each letter in a separate layer. It also puts the period over the ‘i’ in a separate layer, so we will move that into the layer the ‘i’ is in and then delete the empty layer.

Each letter is converted into a symbol. I then created two motion guides. The first for the word String, and the second for the word Along. I draw a curvy line in each guide and each letter is placed on the line in an off-screen position and finally in their proper title position. Each letter then has motion tweening applied to it.

Figure 2 String Along Title Screen with motion guides shown

The last thing we have to do is create buttons for the five different speeds that we have. Buttons will be simply a rectangle that changes colors based on the state with text over it. With the five buttons created, we place them on the screen and have them slide in at intervals. To activate the buttons we need the following code

```
slowest_btn.onRelease = function()
{
      speed = 1;
      speed_setting = 1;
      gotoAndPlay("Game", 1);
};

slow_btn.onRelease = function()
{
      speed = 2;
      speed_setting = 2;
      gotoAndPlay("Game", 1);
};

normal_btn.onRelease = function()
{
      speed = 4;
      speed_setting = 3;
      gotoAndPlay("Game", 1);
};

fast_btn.onRelease = function()
{
      speed = 8;
      speed_setting = 4;
      gotoAndPlay("Game", 1);
};

fastest_btn.onRelease = function()
{
      speed = 16;
      speed_setting = 5;
      gotoAndPlay("Game", 1);
};
```

And finally, we remove the following two lines from the games initialization

```
speed = 2;
speed_setting = 2;
```