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

Chapter 32

Lights Out

Contents

Now we are ready for a bit of smashing action, so we create our Lights Out game, which relies heavily on trigonometry and a bit of physics.

Designing the Lights and Racket

The first thing that we are going to have to decide for the Lights Out game is the frame rate of the game. For many of the other games, having a frame rate of 12 frames per second was more than adequate for the game. Arcade games, however, tend to have much more going on, so a higher frame rate is pretty much a necessity. I would go with 15, 20, or 30 frames per second. The best course is to go for the larger 30 frames per second and reduce it only if there are problems.

Now we can start creating the assets that the game will use. Let us start with the player and the instigator ball. While the rough artwork could be done in Flash, I used fireworks to play around with possible shapes for the player.

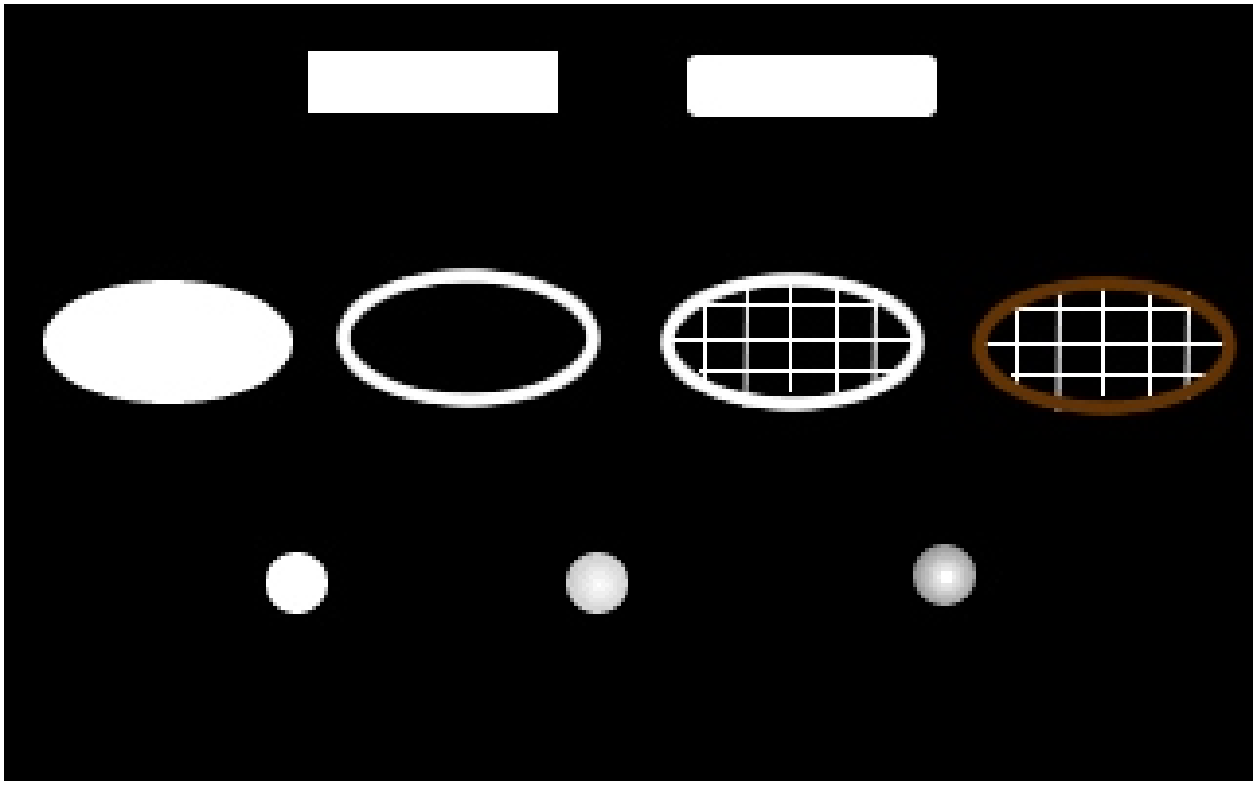

Figure 1: Ball and Racket design

Originally I started with your traditional pong paddle. I rounded off the corners but that still didn't give me the look I wanted. I then went with an oval shape. This gave me the idea of turning the player's paddle into something that actually looked like a racket. This was done by adding string and then colouring the oval to make it look more like wood.

The instigator ball also serves as a template for the colored lights. This is because the colored lights are essentially larger versions of the instigator ball. I stared with a flat ball. To that I switched to a radial fill to give me more of a three-dimensional look for the ball. Finally I increased the size of the glow spot by adding another white and a lighter gray area in the gradient.

The light is a movie which we break into seven labelled values, with the labels being the colors. Each labelled frame has a light image, with the light having different gradient colors. Figure 2 shows the lights with the gradient settings used to create them.

Figure 2: Light Gradients

Coding for Color

Obviously we are going to need a way to change the color of the ball. At the same time we are going to want some easy to remember color constants. The constants can be defined in an initialization phase. This can easily be written to run only once. In fact, we can also make the constants global and make sure new instances of the Ball movie don't re-initialize the variables as follows.

```
init_ball();
stop();

// initialize only once, setting up global constants
function init_ball()
{
      if (ball_initialized != undefined)
            return;
      ball_initialized = true;
      previousBall = null;
      nextBall = null;
      lasthit = 0;
      if (_global.BALL_WHITE == 7)
      {
            return;
      }

      _global.BALL_RED = 1;
      _global.BALL_GREEN = 2;
      _global.BALL_BLUE = 3;
      _global.BALL_MAGENTA = 4;
      _global.BALL_YELLOW = 5;
      _global.BALL_CYAN = 6;
      _global.BALL_WHITE = 7;
      _global.BALL_RAND = new Math.random();
}
```

Now, to set the color of the ball we simply write a function which will change the frame to the appropriate color. To make sure this works, we should also put stop()'s in the last frame of all the different sections.

```
function setColor(n)
{
	current_color = n;
	_alpha = 100;
	switch (n)
	{
		case BALL_RED:
			gotoAndPlay("Red");
			break;
		case BALL_GREEN:
			gotoAndPlay("Green");
			break;
		case BALL_BLUE:
			gotoAndPlay("Blue");
			break;
		case BALL_MAGENTA:
			gotoAndPlay("Magenta");
			break;
		case BALL_YELLOW:
			gotoAndPlay("Yellow");
			break;
		case BALL_CYAN:
			gotoAndPlay("Cyan");
			break;
		case BALL_WHITE:
			gotoAndPlay("White");
			break;
	}
}
```

Some Paddling Action

The first thing that I have to do is import the images for the racket and instigator ball that I created in fireworks into the game. As Macromedia makes both products, this is simply a matter of cutting and pasting. I then need to take the two movie symbols I created and make sure that the reference point is in the center of these objects. I also rename them Instigator and Racket.

At this point we want to get the racket to be tied with movement of the mouse. For this, we move the racket to the bottom of the window. We then write the following code in the newly created code layer of the main movie.

```
init_game();

function onMouseMove() {
      var nx;
      nx = _root._xmouse;
      if (nx < PLAYER_LEFT_BOUND)
            nx = PLAYER_LEFT_BOUND;
      else if (nx > PLAYER_RIGHT_BOUND)
            nx = PLAYER_RIGHT_BOUND;
      player_movie._x = nx;
};

// sets up game constants
function init_game()
{
      if (gameInitialized != undefined)
            return;

      gameInitialized = true;
      PLAYER_LEFT_BOUND = -30;
      PLAYER_RIGHT_BOUND = 640 + 30;
}
```

Quite simply, it sets up a mouse listener that moves the racket based on the position of the mouse. Later, if we desire, we can also hide the mouse cursor.

A Fancy Layout Sequence

First, we need to add a bit to the initialization of the game. This code is added to the initialize function. Two new constants for placement of the playfield are created. These are here so that we can easily adjust the placement of the lights if we need to. Next, we create an array of rows of lights. Each element of this array will hold another array which will happen to hold the lights that are still remaining in that row.

```
	LIGHT_ROWSTART = 72;
	LIGHT_COLUMNSTART = 16;

	light_rows = new Array(7);
	for (var cntr = 0; cntr < 7; ++cntr)
	{
		light_rows[cntr] = new Array();
	}
```

On frame 5 we create a keyframe and label it “startLevel.” This will have a bit of code that will clear out the arrays (in case the game is being re-started).

```
currentRow = 1;
currentColumn = 1;

for (var cntr = 0; cntr < 7; ++cntr)
{
	light_rows[cntr].splice(0, light_rows[cntr].length);
}
```

Now that we are starting the level, we want the lights to appear on the screen. By creating a bit of a loop we can have the lights appear one at a time. To do this, we create a keyframe (on frame 6 or 7) which we will call “layoutLoop.” The code in this frame is where the cloned movies are created. All the movies are clones of a light movie which we drag underneath the visible screen and label “light_movie.”

```
oldRow = currentRow;
bname = "b_"+currentRow+"_"+currentColumn;
light_movie.duplicateMovieClip(bname, currentRow*20+currentColumn);
```

Finally a couple of frames after layoutLoop, we have the code that sets up the balls position and color then completes the loop. The number of frames after the layoutLoop frame is really based on how quickly you want the lights to appear on the screen. Note that I have a trace statement for when the sequence to aid in debugging if there are problems.

```
temp = eval(bname);
temp._x = 32*currentColumn-32+LIGHT_COLUMNSTART;
temp._y = 32*currentRow-32+LIGHT_ROWSTART;
temp.setColor(8-currentRow);
light_rows[currentRow-1].push(temp);
++currentColumn;
if (currentColumn > 20)
{
      currentColumn = 1;
      ++currentRow;
}

if (currentRow < 8)
      gotoAndPlay("layoutLoop");
else
      trace("Sequence done");
```

Now we have a nice movie that lays out the playfield.

Dropping the Ball

We are ready to add the instigator ball into the game. The speed of the ball will be based on the level the player is on, so we are going to have to add support for tracking the current level. This is simply done by adding a variable to the top of the first frame.

```
current_level = 1;
```

Next we need to set up some constants for the instigator. This code is placed in the games initialization function. We are using constants for this information so that when we are fine tuning the game all the information is in one easily accessible place.

```
      INSTIGATOR_START_X = 320-8;
      INSTIGATOR_START_Y = 220;
      INSTIGATOR_TOP = 8;
      INSTIGATOR_LEFT = 8;
      INSTIGATOR_RIGHT = 640-8;
      INSTIGATOR_BOTTOM = 480+8;
      INSTIGATOR_BASESPEED = 1;
      INSTIGATOR_PERLEVELSPEED = 2;
      INSTIGATOR_MAX_SPEED = 12;
      PLAYER_Y = 440;
```

Now, we want to add a new label to our timeline, which we will call "DropInstigator". This will have the code for resetting the ball when it is lost.

```
instigator_movie._x = INSTIGATOR_START_X;
instigator_movie._y = INSTIGATOR_START_Y;
instigator_alive = true;
instigator_angle = Math.PI * 1.5;
instigator_speed = current_level * INSTIGATOR_PERLEVELSPEED +
INSTIGATOR_BASESPEED;
if (instigator_speed > INSTIGATOR_MAX_SPEED)
      instigator_speed = INSTIGATOR_MAX_SPEED;
```

Notice that for the angle we are using radians. This is because the math functions in Flash (and in most other programming languages) use radians. While we could convert the radians into angles, but once you get use to radians, it is just as easy to use radians and would actually be a slight bit faster.

To be honest, I use always use degrees and convert radians into degrees, but when I started playing around with ray casting, I discovered that degrees didn't work very nice. Instead, I had to use my own circle that had 1600 units in a circle. Other ray casters, such as those used in Coffee Quest (even though Coffee Quest 1-4 are step based, I still used ray casting in CQ3 and 4 for generating the maze). For me, the best way of thinking about the subject is that a circle is 2PI radians which can be converted into a circle of however many units is desired.

When the ball finally does appear, we want to give the player a chance to re-orient themselves to the game. To do this we give the ball a little hover time before we get to the main game play loop, which is labelled "MainLoop". In this loop we adjust the position of the instigator (and later we will add code to move falling balls). This is a very simple calculation of the offset of the particular angle. The sin is negative because computer coordinates are different from Cartesian coordinates.

```
var xadj = Math.cos(instigator_angle) * instigator_speed;
var yadj = -Math.sin(instigator_angle) * instigator_speed;
instigator_movie._x += xadj;
instigator_movie._y += yadj;
```

Finally, in the frame after the main loop, we do any additional work (which at the moment is nothing) before looping back to the MainLoop. I should point out at this time that looping is not necessary. Instead, you could simply use the onEnterFrame function to do your main game loop work. Still, it is always nice to have alternatives and when I originally wrote this game, I used a loop so I am going to stick with it.

Bouncing

Right now the instigator ball just drops off the screen. What we need is some bouncing. First, let's worry about bouncing off the four walls (even the bottom wall for now, though later this will result in loss of a ball). This is going to need some new constants which we will add to the game initialization function.

```
	TWOPI = 2 * Math.PI;
	HLIMIT_RIGHT_LOW = Math.PI / 180 * 5;
	HLIMIT_RIGHT_HIGH = TWOPI - HLIMIT_RIGHT_LOW;
	HLIMIT_LEFT_LOW = Math.PI - HLIMIT_RIGHT_LOW;
	HLIMIT_LEFT_HIGH = Math.PI + HLIMIT_RIGHT_LOW;
```

The computer doesn't know how to bounce an object around so we will add a bounce function. This function is based on the fact that the angle of incident is equal to the angle of reflection. By using the provided surface normal we can then calculate the proper angle. By requiring a surface normal, we can have much greater control over allowable angles.

In English, the above paragraph is saying that we will base the angle the instigator bounces of an object on the angle that the instigator hits the object. The angle the object hits another object is based on the angle that the object being hit is oriented to. To reflect this information, we use something called a surface normal. For this program, this just happens to be a radian indicating what angle a straight on hit will bounce. You will also notice that we make sure that you do not bounce off the sides at a horizontal angle (which would lead to a situation where the ball was stuck bouncing back and forth with the player unable to do anything).

```
function bounce(n)
{
	var newAngle = n - instigator_angle + (n - Math.PI);
	while (newAngle > TWOPI)
		newAngle -= TWOPI;
	while (newAngle < 0)
		newAngle += TWOPI;

	// make sure doesn't get stuck horizontally
	if (newAngle < HLIMIT_RIGHT_LOW)
		newAngle = HLIMIT_RIGHT_LOW;
	if (newAngle > HLIMIT_RIGHT_HIGH)
		newAngle = HLIMIT_RIGHT_HIGH;
	if ((newAngle > HLIMIT_LEFT_LOW) && (newAngle < HLIMIT_LEFT_HIGH))
		newAngle = HLIMIT_LEFT_HIGH;

	instigator_angle = newAngle;
}
```

Finally we replace the code on the frame right after the MainLoop label with the following code. This is what determines if the instigator has hit a wall.

```
if (instigator_movie._y < INSTIGATOR_TOP)
{
      instigator_movie._y = INSTIGATOR_TOP;
      bounce(Math.PI * 1.5);
}

if (instigator_movie._x < INSTIGATOR_LEFT)
{
      instigator_movie._x = INSTIGATOR_LEFT;
      bounce(0);
}

if (instigator_movie._y > INSTIGATOR_BOTTOM)
{
      instigator_movie._y = INSTIGATOR_BOTTOM;
      bounce(Math.PI * 0.5);
}

if (instigator_movie._x > INSTIGATOR_RIGHT)
{
      instigator_movie._x = INSTIGATOR_RIGHT;
      bounce(Math.PI);
}

gotoAndPlay("MainLoop");
```

Now you can see a ball bouncing around the playfield. It doesn't interact with anything other than the walls, but at least it is bouncing.

Bouncing off Rackets and Lights

While bouncing off the walls is certainly a necessary feature, it is not much of a game if the player can not control the ball. We already have a racket that the player can control, so now we need code to handle bouncing the instigator ball off the player's racket. To give the player a bit more control of the ball, the angle will be dependent on where the ball hits the racket. This is easily done by varying the normal. Of course, this code is added to the frame after the mainLoop frame.

```
if ( (instigator_movie._y >= PLAYER_Y) &&
                  (instigator_movie._y < (PLAYER_Y+16)) &&
                  (instigator_movie._x > (player_movie._x - 40)) &&
                  (instigator_movie._x < (player_movie._x + 40)) &&
                  (instigator_angle > Math.PI) &&
                  (instigator_angle < TWOPI) )
     bounce(Math.PI * (0.5 + (player_movie._x - instigator_movie._x)/200));
```

Now comes the dreaded hitting the lights part of the game. The first thing we are going to need is a way of calculating an angle between two points. This will allow us to determine the normal based on which angle the instigator hit the ball. While my original plans was to use this value, there is a bit of a problem with doing this, as we will discuss shortly. First, let's write a function for finding the angle.

The first step in finding the angle is determining the distance between the points measured along both the X and Y access. In other words, how far along the X axis would you have to move to get from x1 to x2, and how far along the Y axis would you have to move to get from point y1 to point y2. We will want the absolute value (non-signed) for both of these measurements.

Once we have these measurements, we can find the tangent of the resulting triangle that would be formed if we were to draw a triangle using the angle and the x and y axis. For those not up on their trigonometry, the tangent would be calculated using the absolute value of Y's distance divided by the absolute value of X's distance. There is a potential hazard if the angle is 90 or 270 degrees as in those cases, X is 0. This is easy enough to check for. The arc tangent of the tangent is then the proper angle. Sort of. Arc Tangent only returns values within a certain range. So the bulk of the function is taking the arc tangent result and figuring out the actual angle based on the direction the distances are (as they can be positive or negative, with four combinations putting the angle in one of the four quadrants of the circle).

```
function findAngle(x1, y1, x2, y2)
{
	var distX = x2 - x1;
	var distY = y2 - y1;
	var absX = Math.abs(distX);
	var absY = Math.abs(distY);
	if (absX == 0)
	{
		if (distY > 0)
			return (Math.PI * 1.5);
		else
			return (Math.PI / 2);
	}
	var tang = absY / absX;
	var atang = Math.atan(tang);
	if (distX < 0)
	{
		if (distY < 0)
		{
			return (Math.PI - atang);
		}
		else
		{
			return (Math.PI + atang);
		}
	}
	else
	{
		if (distY < 0)
		{
			return (atang);
		}
		else
		{
			return (TWOPI - atang);
		}
	}
}
```

The next step is to actually see if the instigator has hit one of the balls. When the field is full, there are 140 lights. That is an awful lot of objects to check against. Especially since we know that in order to hit a light the ball needs to be near that row. This means that if we check to see if the ball is within potential striking distance of a row, we can skip at least 5 rows!

Once we know that the instigator can actually potentially hit a ball do we do the collision check. I am using a variation of the bounding circle collision detection. What this does is looks at the distance squared between two point. Recall that the length of a line is the square root of the distance along the X axis squared added to the distance along the Y axis squared. As square roots are very time consuming, the squared distance is just your normal distance calculation but with the square root removed.

I am sure some of you are wondering what good this does? As both objects are round, we know that the distance from the radius to the outer edge of the object is the square root of r*r*2. Therefore, if one of the objects is intersecting the other, the distance between those objects would be less than $2(r_1+r_2)^2$. As we know the instigator's radius and the light's radius, we then know that the distance has to be less than $2(8+16)^2$. For those of you without calculators, that is 1152. This information should be placed in constants, but I'm in a bit of a rush so am using straight numbers.

```
var nxt, temp, rowy1, rowy2, distx, disty, dist, cntrLight;
for (var cntr = 0; cntr < 7; ++cntr)
{
      rowy1 = 32 * cntr - 24 + LIGHT_ROWSTART;
      rowy2 = 32 * cntr + 24 + LIGHT_ROWSTART;
      if ( (instigator_movie._y > rowy1) && (instigator_movie._y < rowy2) )
      {
            trace ("Possible collision with row " + cntr);
            for (cntrLight = 0; cntrLight < light_rows[cntr].length;
++cntrLight)
            {
                  temp = light_rows[cntr][cntrLight];
                  distx = temp._x - instigator_movie._x;
                  disty = temp._y - instigator_movie._y;
                  dist = distx * distx + disty*disty;
                  if (dist < 1152)
                  {
                        var norm = findAngle(temp._x, temp._y,
instigator_movie._x, instigator_movie._y);
                        trace("hit occured...." + norm + " y=" +
instigator_movie._y+ " ball y=" +temp._y);
                        instigator_angle = getValidAngle(Math.PI - norm);
                        instigator_movie._x = temp._x + (Math.cos(norm) * 24);
                        instigator_movie._y = temp._y - (Math.sin(norm) * 24);
                        break;
                  }
            }
      }
}
```

Finally, you will notice that this code makes a call to a function called getValidAngle. I wrote this function to solve any problems with the angle of the instigator not being between 0 and 2 PI.

```
function getValidAngle(n)
{
      newAngle = n;
      while (newAngle > TWOPI)
            newAngle -= TWOPI;
      while (newAngle < 0)
            newAngle += TWOPI;
      return newAngle;
}
```

Smashing Lights

The instigator ball now bounces off lights, but we want the lights to break. More particularly, if the light is a non-prime number, we want one of the colors to be knocked off of the light. To handle this, we add a new function to the light movie that will change the color for us and returns true if the light has been destroyed.

```
function splitLight()
{
	var rremove = false;
	var hitdate = new Date();
	var hittime = hitdate.getTime();
	trace (hittime + " " + lasthit);
	if ((hittime - lasthit) < 100)
		return false;
	lasthit = hittime;
	var rnd;
	switch (current_color)
	{
		case BALL_RED:
		case BALL_GREEN:
		case BALL_BLUE:
			rremove = true;
			break;
		case BALL_MAGENTA:
			rnd = Math.floor(Math.random()*2);
			if (rnd == 1)
				setColor(BALL_BLUE);
			else
				setColor(BALL_RED);
			break;
		case BALL_YELLOW:
			rnd = Math.floor(Math.random()*2);
			if (rnd == 1)
				setColor(BALL_GREEN);
			else
				setColor(BALL_RED);
			break;
		case BALL_CYAN:
			rnd = Math.floor(Math.random()*2);
			if (rnd == 1)
				setColor(BALL_BLUE);
			else
				setColor(BALL_GREEN);
			break;
		case BALL_WHITE:
			rnd = Math.floor(Math.random()*3) + BALL_MAGENTA;
			setColor(rnd);
			break;
	}
	return rremove;
}
```

We are going to need to have a list of dead lights (eventually, we are going to have to destroy all the light movies) so I add a new variable to the initialization function.

```
      dead_lights = new Array();
```

For now, we will let the garbage collection handle the code removal. Later on we will write a better cleaning function. To clear the list we add this line of code to the StartLevel frame.

```
dead_lights.splice(0, dead_lights.length);
```

Finally we add the code for handling the lights splitting or breaking. This code is placed in the true portion of the collision detection code we wrote (part of the if (dist < 1152) code block) just before the break statement. It simply calls the light's splitLight funtion which will return true if the light needs to be removed. To remove the light, we first dim the light by reducing it's alpha level. Then we add the light to the dead list and remove it from the row array. To give you a better idea of what is happening, I added some trace statements.

```
                        if (temp.splitLight())
                        {
                              temp._alpha = 30;
                              dead_lights.push(temp);
                              light_rows[cntr].splice(cntrLight, 1);
                              trace ("Row " + cntr + " has " +
light_rows[cntr].length + " elements.");
                              trace ("dead_lights has " + dead_lights.length +
" elements.");
                        }
```

Now you can see a pretty functional game. Lights are destroyed and the player can control the paddle. Still, we have a bit of work to do to finish the game.

Clearing Levels

At this point we have quite a nice game. What we need to do now is handle the clearing of a level. When you think about it, a level is clear when the dead light list has 140 lights in it (seven rows of twenty).

To implement this, we first go to the frame after MainLoop and remove the line:

```
gotoAndPlay("MainLoop");
```

At the end of the code we add a slightly modified version of this line as follows:

```
if (dead_lights.length >= 140)
      gotoAndPlay("LevelComplete");
else
      gotoAndPlay("MainLoop");
```

Finally, we create the frame labelled “LevelComplete”. To hold special messages, such as the one we are about to show, we create a new layer which I called “messages.” Within this new layer we use a big font and write the words “Level Complete” on it. I let it run for thirty frames (remember we are running at thirty frames per second). Finally on the last frame of the sequence we have the following action script.

```
++current_level;
GotoAndPlay("StartLevel");
```

Scoring

Now we are ready to add scoring to the game. There are 4 elements of the score line that we are going to track. The level, lives, high score, and the score. Only one of these has a variable associated with it. So, let's create variables to hold the other three values. In the initGame function add the following lines

```
      current_score = 0;
      high_score = 0;
```

At the start of the frame we also add the following

```
current_lives = 3;
if (current_score > high_score)
      high_score = current_score;
current_score = 0;
```

Now we need to create a layer to hold the four display variables. These are just dynamic text blocks which I named "lives_txt", "level_txt", "high_txt" and "score_txt". To give them up to date values, I added the following to the MainLoop

```
lives_txt.text = "LIVES " + current_lives;
level_txt.text = "LEVEL " + current_level;
high_txt.text = "HIGH SCORE " + high_score;
score_txt.text = "SCORE " + current_score;
```

And to actually implement the scoring, in the loop for checking the ball I added the following line into the block of code for handling balls being hit. The line was placed at the beginging of the code block that begins with “if (temp.splitLight())”

```
current_score = current_score + (7-cntr) * 10;
```

In other words, you gain points only when you have destroyed a light, and the points scored are based on which row the light is in (with higher rows earning more points).

Ending the Game

The game will now run forever, as there currently is no way of losing the game. In order for the player to lose, they must lose all the instigator balls. Right now that is not possible as the ball bounces when it hits the bottom of the screen. It is a fairly simple to change this so that. First, change the lines in the frame after MainLoop from

```
if (instigator_movie._y > INSTIGATOR_BOTTOM)
{
      instigator_movie._y = INSTIGATOR_BOTTOM;
      bounce(Math.PI * 0.5);
}
```

to

```
var lostInst = false;
if (instigator_movie._y > INSTIGATOR_BOTTOM)
{
      --current_lives;
      lostInst = true;
      if (current_lives <= 0)
            gotoAndPlay("GameOver");
      else
            gotoAndPlay("DropInstigator");
}
```

While you would think this would be enough, remember that the gotoAndPlay function only changes the frame once the script has finished executing. In other words, if another gotoAndPlay is encountered before the end of the script (and there is one) then that will become the frame which Flash goes to. Since we don't want Flash to change it's mind about where it is going to branch to (which it will), we will change the last lines of the frame to

```
if (lostInst)
      ;
else if (dead_lights.length >= 140)
      gotoAndPlay("LevelComplete");
else
      gotoAndPlay("MainLoop");
```

Now, we need a game over sequence. This could be just a simple text message, and in fact, for the most part that is all that it is. I just feel that there needs to be some type of feeling of finality. To achieve that, I am going to make the letters appear slowly.

To start with, I am going to create a frame labelled GameOver. I am then going to add a layer above the message layer which I will call AnimText. In this layer I will write the GAME OVER message. I will then break the message appart and save G, A, M, E, and OVER as five separate

symbols.

For each of the letters, I will have the letter fade in over 20 frames. Each letter will appear separatly and once the letter has appeared, it will be moved to the message layer so the animText layer will be free for the next letter.

Once all the letters have appeared, that word OVER will appear. This will not be a fade in, but will instead be instantly appearing for a bit of impact. Leave a couple of seconds (60 frames) for the message to get accross to the player and then on the last frame have the following code.

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

On the first frame of the sequence I have the following code:

```
lives_txt.text = "LIVES 0";
clearLights();
```

Which of course is there to solve the problem of the balls covering up the game over sign. Of course, we need to write a routine that destroys all the light movies, which is below:

```
function clearLights()
{
	var cntrLight;

	for (var cntr = 0; cntr < 7; ++cntr)
	{
		for (cntrLight = 0; cntrLight < light_rows[cntr].length;
++cntrLight)
		{
			removeMovieClip(light_rows[cntr][cntrLight]);
		}
		light_rows[cntr].splice(0, light_rows[cntr].length);
	}

	for (cntrLight = 0; cntrLight < dead_lights.length; ++cntrLight)
	{
		removeMovieClip(dead_lights[cntrLight]);
	}
	dead_lights.splice(0, dead_lights.length);
}
```

Observant readers will realize that this is an extended version of the code on the StartLevel frame and are probably wondering why we need all the code on the start level frame if this function will do the job better? The answer is that we don't. Here is the code that replaces all the code on the StartLevel frame.

```
currentRow = 1;
currentColumn = 1;
```

```
clearLights();
```

The Title Screen

The title screen needed to reflect the game. To do this, I thought of having a growing spotlight with LIGHTS being in the various light colors and OUT being black. Designing this image was very simple. I started with a circle and then wrote the word LIGHTS in a big font. Broke it apart twice to turn the letters into objects. I colored the letters and placed them in the appropriate locations in the circle. I did the same for OUT, except in that case it was colored black. The title screen is shown below.

Figure 3: Title Screen

The starting animation is a simple size tween. On the first frame we have a small version of the logo. On frame 60 we have the ending size. The tween lasts 60 frames because we are running at 30fps.The start button is just the words “Start the Game” converted to a button. The code to run this is very simple.

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

Fine Tuning

So far good, but there are a few things to do to get the game just right. First, all arcade games need decent sound. Especially this type of game. Importing sound files is simple enough. Once imported, you need to adjust the sound clip's linkage. This should be set to "Export for Action Script" so that Action Script can see the sound symbol. Then, in the initialization frame you can set sound objects for each of the four sounds.

```
breakSound = new Sound(this);
breakSound.attachSound("glass");
bounceSound = new Sound(this);
bounceSound.attachSound("bounce");
levelSound = new Sound(this);
levelSound.attachSound("levelUp");
fireSound = new Sound(this);
fireSound.attachSound("fireball");
```

The break sound is placed where collision detection with the lights occurs just before or after the score is updated. The code is just the following.

```
breakSound.start();
```

The bounce sound can conveniently be placed in the bounce function, as follows:

```
bounceSound.start();
```

The level sound can be placed on the first frame of the "LevelComplete" section. If you haven't guessed already, the code is

```
levelSound.start();
```

Finally, when the player loses an instigator, we place the final bit of code.

```
fireSound.start();
```

Next, let us adjust the starting location of the lights and paddle and then speed up the ball. This is just a mater of adjusting the constant values we set up in the initialization section of the code. Just find and change these variables.

```
LIGHT_ROWSTART = 122;
LIGHT_COLUMNSTART = 16;
INSTIGATOR_START_Y = 350;
INSTIGATOR_BASESPEED = 2;
INSTIGATOR_PERLEVELSPEED = 2;
```

Now playing the game is better but the instigator seems to be bouncing funny. After looking at the code for handling the instigator bouncing off of the lights, I noticed a problem. The value of the bounced angle should actually be set to norm. This mistake was probably made because of the tight schedule I am currently working under. While putting in long hours may seem more productive, if you do it for more than a couple weeks at a time you start making dumb mistakes. Actually, that's not quite true. I should say that you tend to make more dumb mistakes than you normally do, as I am sure every one occasionally makes dumb mistakes (mine generally just happen to be with who I choose to date, but this book is not the place for that).

The game is a bit nicer but it is kind of hard to figure out when the ball is going to bounce of a wall. The solution, draw a box around the screen.

Finally, the start button could be a bit better. I simply added a background box underneath the text that is invisible in the "up" frame, blue in the "over" frame, and red in the "down" frame.

Now all that is needed is to export the game and you have a finished game.