Combining Sound with Motion and Image Animation*

R.G. (Dick) Baldwin

Abstract

Learn to combine sounds, motion animation, image animation, and other interesting effects in a Flash movie using ActionScript 3.

NOTE: **Click** LighteningStorm01 [1] to run the ActionScript program from this lesson. This program produces sound in addition to graphics. *(Click the "Back" button in your browser to return to this page.)*

1 Table of Contents

*Version 1.2: Jun 6, 2010 1:25 pm -0500
†http://creativecommons.org/licenses/by/3.0/
[1] http://cnx.org/content/m34498/latest/LighteningStorm01.html

2 Preface

2.1 General

This tutorial lesson is part of a series of lessons dedicated to object-oriented programming *(OOP)* with ActionScript.

The project that I will present and explain in this lesson is the culmination of several previous lessons dealing with animation, sound, transparency, mouse events, chroma key, etc.

NOTE: All references to ActionScript in this lesson are references to version 3.0 or later.

Several ways to create and launch ActionScript programs

There are several ways to create and launch programs written in the ActionScript programming language. Many of the lessons in this series will use Adobe Flex as the launch pad for the sample ActionScript programs.

An earlier lesson titled **The Default Application Container** provided information on how to get started programming with Adobe's Flex Builder 3. The lesson titled **Using Flex 3 in a Flex 4 World** was added later to accommodate the release of Flash Builder 4. *(See Baldwin's Flex programming website* [2] *.)* You should study those lessons before embarking on the lessons in this series.

Some understanding of Flex MXML will be required

I also recommend that you study all of the lessons on Baldwin's Flex programming website [3] in parallel with your study of these ActionScript lessons. Eventually you will probably need to understand both ActionScript and Flex and the relationships that exist between them in order to become a successful ActionScript programmer.

Will emphasize ActionScript code

It is often possible to use either ActionScript code or Flex MXML code to achieve the same result. Insofar as this series of lessons is concerned, the emphasis will be on ActionScript code even in those cases where Flex MXML code may be a suitable alternative.

2.2 Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the figures and listings while you are reading about them.

2.2.1 Figures

2.2.2 Listings

[2] http://www.dickbaldwin.com/tocFlex.htm
[3] http://www.dickbaldwin.com/tocFlex.htm

2.3 Supplemental material

I recommend that you also study the other lessons in my extensive collection of online programming tutorials. You will find a consolidated index at www.DickBaldwin.com [4] .

3 General background information

Things you have learned

You learned about event handling, bitmap basics, the fundamentals of image pixel processing, and using chroma key compositing to create transparent backgrounds in earlier lessons in this series.

You also learned about drawing with the **Graphics** class, the fundamentals of animation and using sound in ActionScript in earlier lessons as well.

In the lesson titled **Animation Fundamentals** I told you that I refer to any program code that causes visual images to *change over time* to be animation.

Other types of animation

Although we commonly think of animation in terms of images that appear to move over time, there are many other valid forms of animation as well. For example, if the color of an image changes over time, that is animation. If the transparency of an image changes over time, that is animation. If graphic objects appear and then disappear, that is animation.

In this lesson, I will explain a program that is intended to pull together much of what you have already learned and to introduce you to this broader view of animation as well.

4 Preview

Run the ActionScript program named LighteningStorm01 .

I recommend that you run (p. 1) the online version of this program before continuing.

NOTE: If you don't have the proper Flash Player installed, you should be notified of that fact and given an opportunity to download and install the Flash Player plug-in program.

LighteningStorm01 at startup

The program begins by displaying a scene similar to that shown in Figure 1.

[4] http://www.dickbaldwin.com/toc.htm

LighteningStorm01 at startup.

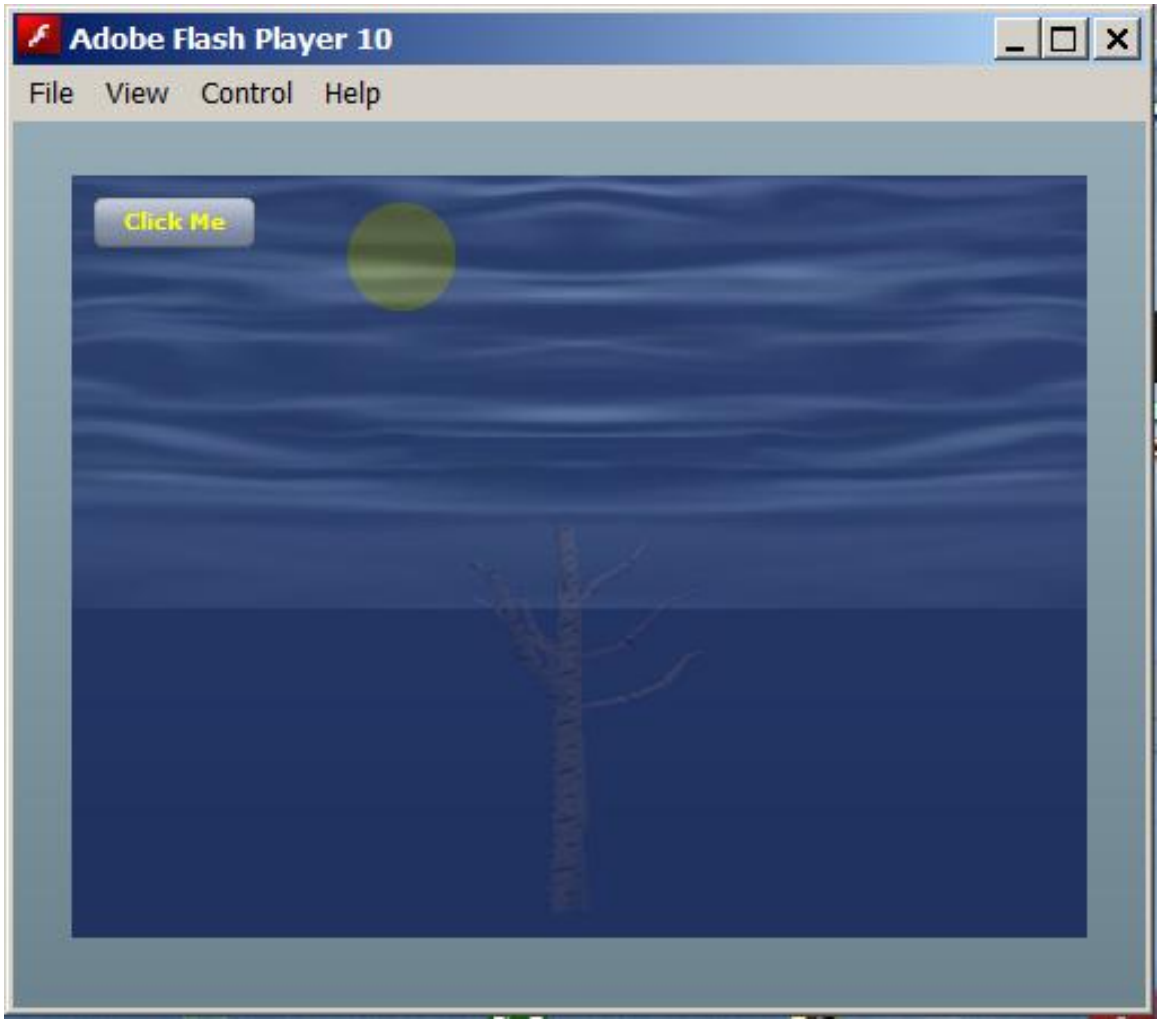

Figure 1: LighteningStorm01 at startup.

What you should see

When the scene in Figure 1 appears, the clouds should seem to be moving slowly. The overall color of the scene should be changing slowly in the bluish-green range. The moon should be moving very slowly from left to right across the screen. You should be able to barely make out a tree in the fog near the bottom center of the image and you should be able to hear the wind and the rain. The sound of the rain should be continuous while the sound of the wind should come and go on a random basis.

Output produced by clicking the button

When you click the button, the scene should change to one similar to that shown in Figure 2.

Visual output produced by clicking the button.

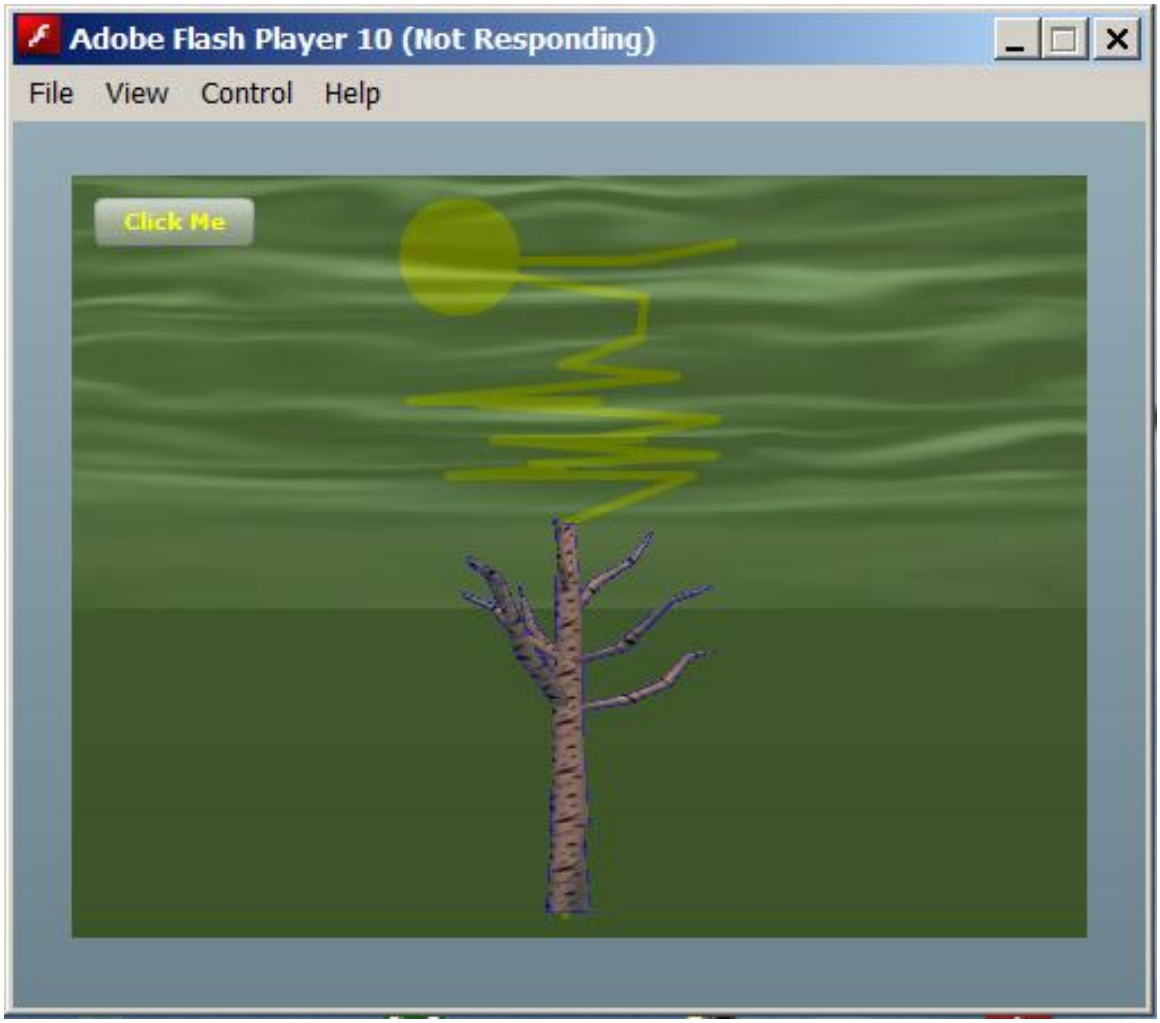

Figure 2: Visual output produced by clicking the button.

The sights and sounds of a lightening bolt

You should hear a sizzle sound as a lightening bolt comes out of the sky and strikes the old tree. The shape of the lightening bolt should be random from one button-click to the next. Except for the motion of the moon and the shape of the tree, the details of all of the visual elements should change over time on in a random fashion.

The flash of the lightening

The lightening bolt should light up the scene with an eerie yellowish-green glow. The overall color of the scene should change slowly and randomly while the sizzle sound is playing and the lightening bolt is visible.

A loud clap of thunder

There should be a loud clap of thunder immediately following the sizzle sound as the scene reverts to

something similar to that shown in Figure 1.

The moon

Throughout all of this, the moon should continue to move very slowly from left to right across the scene. When it reaches the right edge of the scene, it should wrap around and start over on the left side.

5 Discussion and sample code

The project file structure

The final project file structure, captured from the FlashDevelop project window, is shown in Figure 3.

Project file structure for LighteningStorm01.

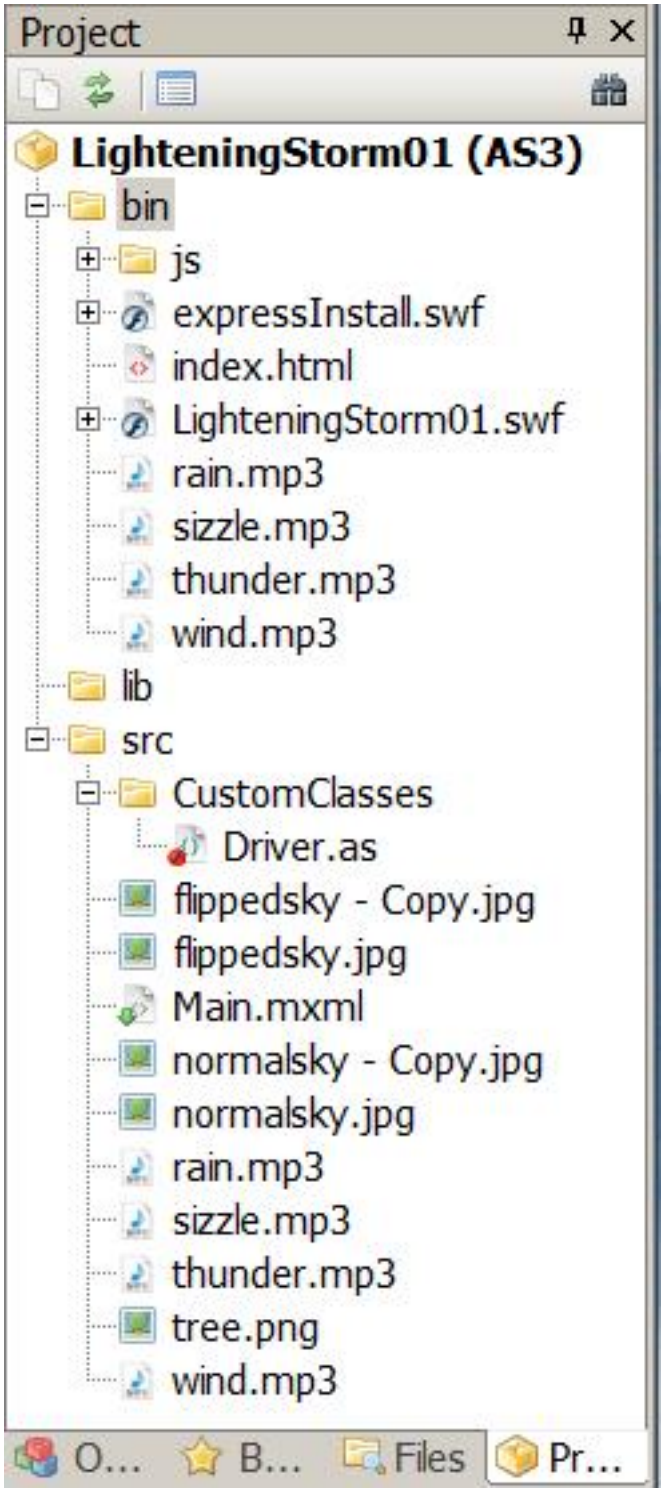

Figure 3: Project file structure for LighteningStorm01.

As you can see in Figure 3, all of the sound and image files are stored in the folder named **src** . In addition, all of the sound files were manually copied into the folder named **bin** .

Will explain in fragments

I will explain the code for this program in fragments. Complete listings of the MXML code and the ActionScript code are provided in Listing 16 and Listing 17 near the end of the lesson.

5.1 The MXML code

The MXML code is shown in Listing 16. As is often the case in this series of lessons, the MXML file is very simple because the program was coded almost entirely in ActionScript. The MXML code simply instantiates an object of the **Driver** class. From that point forward, the behavior of the program is controlled by ActionScript code.

5.2 The ActionScript code

Beginning of the Driver class

The **Driver** class begins in Listing 1.

Listing 1: Beginning of the Driver class.

```
    /*Project LighteningStorm01
This project was developed using FlashDevelop, which
has a couple of requirements that may not exist with
Flex Builder 3 or Flash Builder 4.
1. You must manually copy all mp3 files into the bin
folder.
2. You must insert an extra slash character in the URL
when embedding an image file in the swf file.
************************************************************/
package CustomClasses{
  import flash.display.Bitmap;
  import flash.display.BitmapData;
  import flash.media.SoundChannel;
  import mx.containers.Canvas;
  import mx.controls.Image;
  import mx.controls.Button;
  import mx.events.FlexEvent;
  import flash.events.TimerEvent;
  import flash.events.MouseEvent;
  import flash.utils.Timer;
  import flash.utils.ByteArray;
  import flash.media.Sound;
  import flash.net.URLRequest;
  import flash.media.SoundChannel;
  import flash.events.Event;
  import flash.geom.Rectangle;


  //=====================================================//

  public class Driver extends Canvas {
```

```
//Extending Canvas makes it possible to position
// images with absolute coordinates. The default
// location is 0,0;

private var bkgndColor:uint = 0x005555;
private var redBkgnd:uint = 0;
private var greenBkgnd:uint = 128;
private var blueBkgnd:uint = 128;

private var normalSky:Image = new Image();
private var flippedSky:Image = new Image();

private var tree:Image = new Image();
private var newTreeImage:Image = new Image();
private var treeBitMap:Bitmap;

private var alphaLim:Number = 0.5;
private var normalAlpha:Number = alphaLim;
private var flippedAlpha:Number;
private var normalAlphaDecreasing:Boolean = true;

private var canvasObj:Canvas;
private var timer:Timer = new Timer(35);
private var loopCntr:uint;

private var lighteningCntr:uint = 0;
private var lighteningCntrLim:uint = 25;
private var lighteningStartX:uint;
private var lighteningStartY:uint;
private var lighteningEndX:uint;
private var lighteningEndY:uint;

private var sizzle:Sound;
private var thunder:Sound;
private var wind:Sound;
private var rain:Sound;

private var sizzlePlaying:Boolean = false;
private var channel:SoundChannel;

private var button:Button;

private var radius:Number = 24;//radius of circle
private var circleX:Number = 5 * radius;
private var circleY:Number = 1.5 * radius;
private var dx:Number = 0.05;
```

Nothing new here

There is nothing new in Listing 1, which consists almost entirely of import directives and instance variable declarations, so no further explanation of Listing 1 should be required. I will simply call your attention to the comments regarding the FlashDevelop IDE at the beginning of Listing 1.

Beginning of the constructor for the Driver class

The constructor for the **Driver** class begins in Listing 2.

Listing 2: Beginning of the constructor for the Driver class.

```
  public function Driver(){//constructor
//Make this Canvas visible.
bkgndColor = (redBkgnd ≪ 16) + (greenBkgnd ≪ 8)
                                        + blueBkgnd;
setStyle("backgroundColor", bkgndColor);
setStyle("backgroundAlpha",0.5);
```

A 24-bit color value

The first statement in Listing 2 uses the left bitshift operator to construct a 24-bit binary value that will be used to establish the red, green, and blue values for the initial background color of the **Canvas** object. Hopefully you are already familiar with binary bit shifting. If not, just Google **bitshift operator** and you will find a lot of information on the topic. Note that the left bitshift operator is essentially the same in ActionScript, Java, C++, and other programming languages as well.

Set the initial background color and the transparency

Then Listing 2 calls the **setStyle** method twice in succession to set the background color and the transparency of that color for the background of the canvas.

An examination of the initial values for **redBkgnd** , **greenBkgnd** , and **blueBkgnd** in Listing 1 indicates that the initial background color is a dark shade of cyan with equal contributions of green and blue and no red.

The transparency of the background color

The second call to the **setStyle** method in Listing 2 causes the background color to exhibit a 50-percent transparency or opacity.

It is important that the background color not be completely opaque. If it were opaque, it would not be possible to see the yellow moon and the yellow lightening bolts that are drawn on the canvas behind the background color.

Will change over time

The background color will be changed later in an event handler that is registered on a **Timer** object. The color will not only be subject to small random changes. It will also be subject to major changes switching between the periods when a lightening flash is occurring or not occurring.

The remainder of the constructor

The remainder of the constructor is shown in Listing 3. There is nothing new in Listing 3 so no explanation beyond the embedded comments should be needed.

Listing 3: The remainder of the constructor.

```
    //Load the two sky images and embed them in the
// swf file.
//Note the use of a / to eliminate the "Unable to
// resolve asset for transcoding" Compiler Error
[Embed("/normalsky.jpg")]
var imgNormal:Class;
normalSky.load(imgNormal);

[Embed("/flippedsky.jpg")]
var imgFlipped:Class;
flippedSky.load(imgFlipped);
```

```
      //Load the treeImage and embed it in the swf file.
      [Embed("/tree.png")]
      var imgTree:Class;
      tree.load(imgTree);


      //Load sound files and play two of them.
      sizzle = new Sound();
      sizzle.load(new URLRequest("sizzle.mp3"));

      thunder = new Sound();
      thunder.load(new URLRequest("thunder.mp3"));

      wind = new Sound();
      wind.load(new URLRequest("wind.mp3"));
      wind.play(0,2);//play twice

      rain = new Sound();
      rain.load(new URLRequest("rain.mp3"));
      rain.play(0, int.MAX_VALUE);//play forever


      //Register an event listener on the CREATION_
      // COMPLETE event.
      this.addEventListener(FlexEvent.CREATION_COMPLETE,
                               creationCompleteHandler);

      //Save a reference to this Canvas object, which will
      // be used later for a variety of purposes.
      canvasObj = this;


      //Draw a yellow filled circle on this Canvas object.
      graphics.beginFill(0xffff00);
      graphics.drawCircle(circleX,circleY,radius);
      graphics.endFill();

    } //end constructor
```

The CREATION_COMPLETE event handler

The code in the constructor in Listing 3 registers a **CREATION_COMPLETE** event handler on the **Canvas** object. You are already familiar with the use of event handlers of this type.

The **CREATION_COMPLETE** event handler is shown in its entirety in Listing 4. As before, there is nothing in Listing 4 that I haven't explained in previous lessons, so no explanation beyond the embedded comments should be needed.

Listing 4: The CREATION_COMPLETE event handler.

```
      //This handler method is executed when the Canvas has
    // been fully created.
```

```
    private function creationCompleteHandler(
                          event:mx.events.FlexEvent):void{

      //Set the width and height of the Canvas object
      // based on the size of the bitmap in the
      // normalSky image.
      this.width = Bitmap(normalSky.content).width;
      this.height = Bitmap(normalSky.content).height;

      //Add the images to the Canvas object. Note that
      // the two images are overlaid at 0,0.
      this.addChild(normalSky);
      this.addChild(flippedSky);

      //Add a button in the upper-left corner in front
      // of the sky images and register a CLICK event
      // handler on the button.
      button = new Button();
      button.x = 10;
      button.y = 10;
      button.addEventListener(MouseEvent.CLICK, onClick);
      button.label = "Click Me";
      button.setStyle("color", 0xFFFF00);
      addChild(button);


      //Get and save a reference to a Bitmap object
      // containing the contents of the tree file.
      treeBitMap = Bitmap(tree.content);

      //Place the treeBitMap in a new Image object and
      // place it on the canvas near the bottom center of
      // the canvas.
      treeBitMap.x =
                 canvasObj.width / 2 - treeBitMap.width/2;
      treeBitMap.y = canvasObj.height - treeBitMap.height;

      newTreeImage.addChild(treeBitMap);
      this.addChild(newTreeImage);

      //Make the tree almost invisible. It will be made
      // highly visible in conjunction with a
      // lightening flash.
      newTreeImage.alpha = 0.2;

      //Cause the blue background of the tree to
      // be transparent.
      processChromaKey(treeBitMap);


      //Register a timer listener and start the timer
```

```
    // running.
    timer.addEventListener(TimerEvent.TIMER, onTimer);
    timer.start();

  } //end creationCompleteHandler
```

Beginning of the TIMER event handler

The last two statements in Listing 4 register a **TIMER** event listener on the **Timer** object that was instantiated in Listing 1 and start the timer running.

The **TIMER** event handler begins in Listing 5. There is some interesting new code in this method, so I will break it down and explain it in fragments.

Listing 5: Beginning of the TIMER event handler.

```
    //TimerEvent handler. This method is executed each
  // time the timer object fires an event.
  public function onTimer(event:TimerEvent):void {

    //Update the loop counter. Several things depend on
    // this counter.
    loopCntr++;
    if (loopCntr > int.MAX_VALUE-2) {
      //Guard against numeric overflow.
      loopCntr = 0;
    }//end if

    //Play a wind sound every 100th timer event only
    // if a random value is greater than 0.5. This
    // should happen half the time on the average.
    if ((loopCntr % 100 == 0)&& (Math.random() > 0.5)) {
      wind.play();
    }//end if

    //Make random changes to the background color.
    processBackgroundColor();

    //Make changes to the alpha values of the normal
    // and flipped sky images.
    makeTheCloudsMove();
```

Approximately three Timer events per second

As you are aware, the event handler that begins in Listing 5 is executed each time the **Timer** object fires an event. The **Timer** object was instantiated in Listing 1 and configured to fire an event every 35 milliseconds, or approximately three times per second.

The only thing that is new in the fragment shown in Listing 5 is the pair of calls to the methods named:

- processBackgroundColor, and
- makeTheCloudsMove

The purpose of each of these methods is described by its name. I will explain both methods later in this lesson.

Controlling the motion of the moon

The remainder of the **Timer** event handler, which is shown in Listing 6, is dedicated to causing the moon to move very slowly from left to right across the screen as shown in Figure 1 and Figure 2.

Listing 6: Controlling the motion of the moon.

```
    //Draw a filled circle on this Canvas object.
if (!sizzlePlaying) {
  //Erase the circle. Note that this would also
  // erase the lightening bolt if it were done while
  // the sizzle sound is playing.
  graphics.clear();
}//end if

//Make the circle move a very small distance to the
// right. Make it wrap and reappear on the left
//when it reaches the right side of the window.
circleX += dx;
if (circleX > canvasObj.width - radius) {
  circleX = 5 * radius;
}//end if
graphics.beginFill(0xffff00);
graphics.drawCircle(circleX,circleY,radius);
graphics.endFill();

}//end onTimer
```

The code in Listing 6 draws a yellow filled circle a little further to the right each time the **Timer** fires an event. When the circle reaches the right edge of the Flash window, it starts over again on the left.

Erase the old moon before drawing the new moon

It is necessary to erase the old circle before drawing each new filled circle. Otherwise, instead of seeing a filled circle moving from left to right, the viewer would see a very wide yellow line being slowly drawn across the screen.

Houston, we have a problem

The proper way to erase the old filled circle is to call the **clear** method of the **Graphics** class. However, this is also the proper way to erase the yellow lightening bolt that I will explain later. Therefore, it is critical that the **clear** method not be called while the lightening bolt is on the screen.

The Boolean variable named sizzlePlaying

The **Boolean** variable named **sizzlePlaying** is used to control several aspects of the program relative to the period during which the sizzle sound is played, the lightening bolt is drawn, and the scene is illuminated by the lightening bolt.

The value of this variable is set to false when the variable is declared in Listing 1. It is set to true when the sizzle sound begins playing and is set back to false when the sizzle sound finishes playing. Thus, it is always true while the sizzle sound is playing and is false at all other times.

An egg-shaped moon

The value of **sizzlePlaying** is used in Listing 6 to prevent the **clear** method from being called while the lightening bolt is on the screen. This actually causes the moon to take on a slight egg shape during that period because new versions of the moon are being drawn without erasing the old versions. However, this isn't visually apparent because the moon moves such a short distance during that period. Also, the lightening bolt and not the moon probably commands the attention of the viewer during this period so the distortion isn't very noticeable.

Beyond that, no explanation of the code in Listing 6 beyond the embedded comments should be needed.

The method named processBackgroundColor

As you saw in Listing 5, the **Timer** event handler calls a method named **processBackgroundColor** each time the **Timer** object fires an event *(about three times per second)* . The purpose of the method is to cause the overall color of the image to change slowly over time. The method is shown in its entirety in Listing 7.

Listing 7: The method named processBackgroundColor.

```
    //This method processes the background color. The
// color changes among various shades of cyan when
// there is no lightening bolt. The color changes
// among various shades of dark yellow when there is a
// lightening bolt.
private function processBackgroundColor():void {
  if (!sizzlePlaying) {
    //Vary background color when there is no
    // lightening flash.
    if (Math.random() > 0.5) {
      if (greenBkgnd < 250){
        greenBkgnd += 5;
      }//end if
    }else {
      if(greenBkgnd > 5){
        greenBkgnd -= 5;
      }//end if
    }//end else

    if (Math.random() > 0.5) {
      if (blueBkgnd < 250){
        blueBkgnd += 5;
      }//end if
    }else {
      if(blueBkgnd > 5){
        blueBkgnd -= 5;
      }//end if
    }//end else

  }else {
    //Vary background color during a lightening flash
    if (Math.random() > 0.5) {
      if (greenBkgnd < 245){
        greenBkgnd += 10;
      }//end if
    }else {
      if(greenBkgnd > 10){
        greenBkgnd -= 10;
      }//end if
    }//end else

    if (Math.random() > 0.5) {
      if (redBkgnd < 245){
```

```
          redBkgnd += 10;
        }//end if
      }else {
        if(redBkgnd > 10){
          redBkgnd -= 10;
        }//end if
      }//end else
    }//end else

    bkgndColor = (redBkgnd << 16) + (greenBkgnd << 8)
                                          + blueBkgnd;
    setStyle("backgroundColor", bkgndColor);
    setStyle("backgroundAlpha",0.5);

  }//end processBackgroundColor
```

Long and tedious

The code in Listing 7 is long and tedious but not particularly complicated.

Three sections of code

The code can be broken down into three sections for purposes of explanation. The first section begins at the beginning of the **if** statement and continues down to, but not including the **else** clause. Note that the conditional clause for the **if** statement tests to determine if the value of the variable named **sizzlePlaying** is false.

For a value of false, the code in the **if** statement makes very small random changes to the green and blue components of the background color during those periods when there is no lightening bolt on the screen. The value of the red color component is zero during this period.

The second section

The second section begins with the **else** clause, and the code in this section is executed when the value of **sizzlePlaying** is true.

The code in this section makes very small random changes to the red and green components of the background color during those periods where there is a lightening bolt on the screen. The value of the blue color component is zero during this period.

The third section – apply the color components

The third section of code, consisting of the last three statements, uses the red, green, and blue color component values computed earlier to cause the color of the background to change. Note that this code maintains a 50-percent opacity value for the background color.

The method named makeTheCloudsMove

As you also saw in Listing 5, the **Timer** event handler also calls a method named **makeTheCloudsMove** each time the **Timer** object fires an event, or about three times per second. The purpose of this method is to create the illusion that the clouds shown in Figure 1 and Figure 2 are moving.

The one new thing

The procedure for accomplishing this is probably the only thing in this lesson that I haven explained in one form or another in an earlier lesson.

The image of the clouds shown in Figure 1 and Figure 2 is actually the superposition of two images, one in front of the other. The two images are shown in Figure 4 and Figure 5.

The image named normalsky.jpg.

Figure 4: The image named normalsky.jpg.

The image named flippedsky.jpg.

Figure 5: The image named flippedsky.jpg.

The differences between the images

If you examine these two images carefully, you will see that:

- One is the mirror image of the other.
- One has been given a green tint while the other has been given a magenta tint.

As mentioned earlier, the program displays both of these images, one on top of the other.

The illusion of movement...

The illusion of movement is achieved by causing the alpha transparency value of one image to go down while the alpha transparency value of the other image goes up and vice versa. In other words, the two images are caused to gradually fade in and out in opposition to one another.

Beginning of the method named makeTheCloudsMove

The code that accomplishes this begins in Listing 8.

Listing 8: Beginning of the method named makeTheCloudsMove.

```
    //This method processes the alpha values of the
// normal and flipped sky images.
// The change in alpha values of the overlapping
// images makes it appear that the clouds are
// moving.
private function makeTheCloudsMove():void {

  //Change the decreasing or increasing direction of
  // the changes in the alpha value for the normal
  // sky image when the alpha value hits the limits.
  if (normalAlphaDecreasing && (normalAlpha <= 0.1)) {
    normalAlphaDecreasing = false;
  }else if (!normalAlphaDecreasing &&
                          (normalAlpha >= alphaLim)) {
    normalAlphaDecreasing = true;
  }//end if
```

A saw tooth change in the alpha values

The alpha value for the normal sky image is caused to range from 0.1 to 0.5 in increments of 0.005 in a saw tooth fashion. At the same time, the alpha value for the other image is caused to range between the same limits in an opposing saw tooth fashion.

The code in Listing 8 keeps track whether the alpha values for the normal sky image are going up or going down, and flips the direction whenever the current alpha value crosses one of the limits.

Compute new alpha value for the normal sky

Listing 9 uses that information to compute a new alpha value for the normal sky image.

Listing 9: Compute new alpha value for the normal sky image.

```
    //Increase or decrease the alpha value for the
// normal sky image.
if (normalAlphaDecreasing) {
  normalAlpha -= 0.005;
}else {
  normalAlpha += 0.005;
}//end else
```

Compute new alpha value for the flipped sky image

Listing 10 uses the new alpha value for the normal sky along with the upper limit of the alpha value to compute a new alpha value for the flipped sky. As the alpha value for the normal sky goes up, the alpha value for the flipped sky goes down and vice versa.

Listing 10: Compute new alpha value for the flipped sky image.

```
    //Cause the alpha value for the flipped sky image
// to go down when the value for the normal sky
// image goes up, and vice versa.
flippedAlpha = alphaLim - normalAlpha;
```

Apply the new alpha values to both sky images

Finally, Listing 10 sets the alpha value for each image to the new value.

Listing 11: Apply the new alpha values to both sky images.

```
    //Change the alpha values for both sky images.
  normalSky.alpha = normalAlpha;
  flippedSky.alpha = flippedAlpha;
 }//end makeTheCloudsMove
```

The next time the images are rendered, the new alpha values will be in effect.

The CLICK event handler for the button

The button shown in Figure 1 provides the mechanism by which the viewer can interact with the program.

The code in Listing 4 registers a **CLICK** event handler on the button. Listing 12 shows that **CLICK** event handler. This method is called each time the user clicks the button.

Listing 12: The CLICK event handler for the button.

```
    //This method is a click handler on the button. It
 // causes the lightening flash to occur and the
 // lightening bolt to be drawn.
 private function onClick(event:MouseEvent):void {
   //Don't create another lightening bolt while the
   // previous one is still in progress.
   if(!sizzlePlaying){
     flashLightening();
     drawLightening();
   }//end if
 }//end onClick
```

Create the lightening bolt and its flash

The code in Listing 12 first confirms that the sizzle sound is not currently being played. If not, Listing 12 calls the method named **flashLightening** to illuminate the scene, and calls the method named **drawLightening** to draw the lightening bolt.

The method named flashLightening

The method named **flashLightening** is shown in its entirety in Listing 13.

Listing 13: The method named flashLightening.

```
    private function flashLightening():void {

  //Make the tree more visible. Apparently
  // setting the alpha property has no effect on the
  // alpha byte values that have been individually
  // set. Otherwise, the blue background would
  // become visible.
  newTreeImage.alpha = 1.0;


  //Play a sizzle sound to accompany the flash of
  // lightening. Set a flag to prevent another sizzle
```

```
// sound from being played before this one finishes.
sizzlePlaying = true;
channel = sizzle.play();
//Register an event listener that will be called
// when the sizzle sound finishes playing.
channel.addEventListener(
        Event.SOUND_COMPLETE, soundCompleteHandler);


//Change the background color to a dark yellow.
redBkgnd = 128;
greenBkgnd = 128;
blueBkgnd = 0;

}//end flashLightening
```

Produce the visual and audible effects of the lightening

The purpose of this method is to produce the visible and audible effects of the lightening other than the lightening bolt itself.

The method creates the flash from thelightening bolt, makes the tree more visible during the flash as shown in Figure 2, and plays a sizzle sound that will be followed by a clap of thunder.

Several steps are involved

The method begins by setting the alpha value on the tree image to 1.0 to cause the tree to become totally opaque.

Then it sets the value of the variable named **sizzlePlaying** to true to notify all other parts of the program that a sizzle sound is being played and a lightening bolt is being drawn.

Then it calls the **play** method on the **sizzle Sound** . The **play** method starts the sizzle sound playing and immediately returns a reference to a **SoundChannel** object through which the sound will be played.

The reference to the **SoundChannel** object is saved in the instance variable named **channel** .

Listing 13 registers an event listener on the **SoundChannel** object that will be executed when the sizzle sound finishes playing.

Finally, Listing 13 sets the red, green, and blue background color component values to dark yellow. These values along with the true value of **sizzlePlaying** will be used by the code in Listing 7 to set the background color of the canvas to dark yellow the next time the **Timer** object fires an event.

The drawLightening method

Listing 14 shows the method named **drawLightening** that is called by the **CLICK** event handler on the button in Listing 12 to draw the actual lightening bolt.

Listing 14: The method named drawLightening.

```
  private function drawLightening():void {

lighteningStartX = Math.floor(Math.random()
                              * canvasObj.width / 3)
                              + canvasObj.width / 3;
lighteningStartY =
              Math.random() * canvasObj.height / 10;
lighteningEndX = canvasObj.width / 2 -6;
lighteningEndY =
          canvasObj.height - treeBitMap.height + 10;
```

```
    //Draw a zero width dark yellow line to the starting
    // point of the lightening bolt.
    canvasObj.graphics.lineStyle(0, 0x999900);
    canvasObj.graphics.lineTo(
                   lighteningStartX, lighteningStartY);

    //Set the line style to a bright yellow line that is
    // four pixels thick.
    canvasObj.graphics.lineStyle(4, 0xFFFF00);

    //Declare working variables.
    var tempX:uint;
    var tempY:uint = lighteningStartY;
    var cnt:uint;

    //Use a for loop to draw a lightening bolt with
    // twenty random segments.
    for (cnt = 0; cnt < 20; cnt++ ) {
      //Compute the coordinates of the end of the next
      // line segment.
      tempX = Math.floor(Math.random()
                                * canvasObj.width / 3)
                                + canvasObj.width / 3;
      tempY = tempY + Math.floor(Math.random()
                          * (lighteningEndY - tempY)/5);
      //Draw the line segment.
      canvasObj.graphics.lineTo(tempX,tempY);
    }//end for loop

    //Draw a line segment to the top of the tree.
    canvasObj.graphics.lineTo(
                        lighteningEndX, lighteningEndY);

    //Make the lightening go to ground.
    canvasObj.graphics.lineTo(
        lighteningEndX,
           lighteningEndY + treeBitMap.height - 20);
  }//end drawLightening
```

This method draws a yellow segmented lightening bolt four pixels thick *(as shown in Figure 2)* that is generally random but always ends up striking the top of the tree.

Long and tedious

As was the case earlier, this method is long and tedious but not technically difficult. Therefore, I will leave it as an exercise for the student to wade through the details in order to understand how it draws the lightening bolt.

The method named soundCompleteHandler

That brings us to the method shown in Listing 15 that is called each time a sizzle sound finishes playing.

Listing 15: The method named soundCompleteHandler.

```
  private function soundCompleteHandler(e:Event):void {

//Allow another sizzle sound to be played now that
// this one is finished.
sizzlePlaying = false;
//Play the thunder immediately following the end of
// the sizzle sound.
thunder.play();

//Switch the background color from dark yellow
// to the normal background color.
redBkgnd = 0;
greenBkgnd = 128;
blueBkgnd = 128;

//Erase the lightening bolt. Note that this will
// also erase the yellow circle.
canvasObj.graphics.clear();
//Make the tree almost invisible.
newTreeImage.alpha = 0.2;

}//end soundCompleteHandler
```

Each time this method is called, it sets the **sizzlePlaying** variable to false to clear the way for the sizzle sound to be played again.

Then it plays the thunder sound and sets the color variables so that the background color will be restored to a dark cyan color by the code in Listing 7.

Finally it calls the **clear** method of the **Graphics** class to erase the lightening bolt, which also erases the moon as well.

Then it sets the alpha value on the tree image to a low value to make the tree appear to be lost in the fog.

The method named processChromaKey

That leaves only the method named **processChromaKey** that I haven't explained. The purpose of this method is to cause the blue background pixels of the tree image shown in Figure 6 to become transparent.

The tree image.

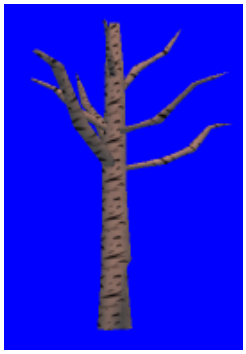

Figure 6: The tree image.

This method is essentially the same as a method that I explained in an earlier lesson titled **Using Chroma Key Compositing to Create Transparent Backgrounds** . Rather than to explain that method again, I will simply refer you to the earlier lesson for an explanation. You can view the method in its entirety in Listing 17.

6 Run the program

I encourage you to run (p. 1) this program from the web. Then copy the code from Listing 16 and Listing 17. Use that code to create your own project. Compile and run the project. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

7 Resources

I will publish a list containing links to ActionScript resources as a separate document. Search for ActionScript Resources in the Connexions search box.

8 Complete program listings

Complete listings of the programs discussed in this lesson are provided below.

Listing 16: Code for Main.mxml

```
    <?xml version="1.0" encoding="utf-8"?>
<!--
Project LighteningStorm01
See Driver.as for a description of this project.
-->

<mx:Application
  xmlns:mx="http://www.adobe.com/2006/mxml"
```

```
  xmlns:cc="CustomClasses.*">

  <cc:Driver/>

</mx:Application>
```

Listing 17: Source code for the class named Driver.

```
    /*Project LighteningStorm01
This project is the culmination of several previous
projects dealing with animation, sound, transparency,
mouse events, chromakey, etc.

When the program starts running, the scene is of a very
stormy day. There is a button in the upper-left corner
of the scene. The clouds are moving. There is also a
yellow moon behind the clouds that is moving very
slowly across the screen from left to right.

When the user clicks the button, a bolt of lightening
comes out of the sky and strikes an image of a tree.
Several aspects of the scene change to reflect
the sights and sounds of a lightening strike.

In addition to the clouds moving, the overall color
of the scene slowly changes randomly. The overall
color varies around a dark cyan when there is no
lightening bolt and varies around a dark yellow when
there is a lightening bolt.

This project was developed using FlashDevelop, which
has a couple of requirements that may not exist with
Flex Builder 3 or Flash Builder 4.
1. You must manually copy all mp3 files into the bin
folder.
2. You must insert an extra slash character in the URL
when embedding an image file in the swf file.
**********************************************************/
package CustomClasses{
  import flash.display.Bitmap;
  import flash.display.BitmapData;
  import flash.media.SoundChannel;
  import mx.containers.Canvas;
  import mx.controls.Image;
  import mx.controls.Button;
  import mx.events.FlexEvent;
  import flash.events.TimerEvent;
  import flash.events.MouseEvent;
  import flash.utils.Timer;
```

```
import flash.utils.ByteArray;
import flash.media.Sound;
import flash.net.URLRequest;
import flash.media.SoundChannel;
import flash.events.Event;
import flash.geom.Rectangle;


//=====================================================//

public class Driver extends Canvas {
  //Extending Canvas makes it possible to position
  // images with absolute coordinates. The default
  // location is 0,0;

  private var bkgndColor:uint = 0x005555;
  private var redBkgnd:uint = 0;
  private var greenBkgnd:uint = 128;
  private var blueBkgnd:uint = 128;

  private var normalSky:Image = new Image();
  private var flippedSky:Image = new Image();

  private var tree:Image = new Image();
  private var newTreeImage:Image = new Image();
  private var treeBitMap:Bitmap;

  private var alphaLim:Number = 0.5;
  private var normalAlpha:Number = alphaLim;
  private var flippedAlpha:Number;
  private var normalAlphaDecreasing:Boolean = true;

  private var canvasObj:Canvas;
  private var timer:Timer = new Timer(35);
  private var loopCntr:uint;

  private var lighteningCntr:uint = 0;
  private var lighteningCntrLim:uint = 25;
  private var lighteningStartX:uint;
  private var lighteningStartY:uint;
  private var lighteningEndX:uint;
  private var lighteningEndY:uint;

  private var sizzle:Sound;
  private var thunder:Sound;
  private var wind:Sound;
  private var rain:Sound;

  private var sizzlePlaying:Boolean = false;
  private var channel:SoundChannel;
```

```
    private var button:Button;

    private var radius:Number = 24;//radius of circle
    private var circleX:Number = 5 * radius;
    private var circleY:Number = 1.5 * radius;
    private var dx:Number = 0.05;
    //-------------------------------------------------//


    public function Driver(){//constructor
      //Make this Canvas visible.
      bkgndColor = (redBkgnd ≪ 16) + (greenBkgnd ≪ 8)
                                              + blueBkgnd;
      setStyle("backgroundColor", bkgndColor);
      setStyle("backgroundAlpha",0.5);

      //Load the two sky images and embed them in the
      // swf file.
      //Note the use of a / to eliminate the "Unable to
      // resolve asset for transcoding" Compiler Error
      [Embed("/normalsky.jpg")]
      var imgNormal:Class;
      normalSky.load(imgNormal);

      [Embed("/flippedsky.jpg")]
      var imgFlipped:Class;
      flippedSky.load(imgFlipped);

      //Load the . treeImage and embed it in the swf file.
      [Embed("/tree.png")]
      var imgTree:Class;
      tree.load(imgTree);


      //Load sound files and play two of them.
      sizzle = new Sound();
      sizzle.load(new URLRequest("sizzle.mp3"));

      thunder = new Sound();
      thunder.load(new URLRequest("thunder.mp3"));

      wind = new Sound();
      wind.load(new URLRequest("wind.mp3"));
      wind.play(0,2);//play twice

      rain = new Sound();
      rain.load(new URLRequest("rain.mp3"));
      rain.play(0, int.MAX_VALUE);//play forever


      //Register an event listener on the CREATION_
```

```
    // COMPLETE event.
    this.addEventListener(FlexEvent.CREATION_COMPLETE,
                              creationCompleteHandler);

    //Save a reference to this Canvas object, which will
    // be used later for a variety of purposes.
    canvasObj = this;


    //Draw a yellow filled circle on this Canvas object.
    graphics.beginFill(0xffff00);
    graphics.drawCircle(circleX,circleY,radius);
    graphics.endFill();

  } //end constructor
  //-------------------------------------------------//

  //This handler method is executed when the Canvas has
  // been fully created.
  private function creationCompleteHandler(
                        event:mx.events.FlexEvent):void{

    //Set the width and height of the Canvas object
    // based on the size of the bitmap in the
    // normalSky image.
    this.width = Bitmap(normalSky.content).width;
    this.height = Bitmap(normalSky.content).height;

    //Add the images to the Canvas object. Note that
    // the two images are overlaid at 0,0.
    this.addChild(normalSky);
    this.addChild(flippedSky);

    //Add a button at in the upper-left corner in front
    // of the sky images.
    button = new Button();
    button.x = 10;
    button.y = 10;
    button.addEventListener(MouseEvent.CLICK, onClick);
    button.label = "Click Me";
    button.setStyle("color", 0xFFFF00);
    addChild(button);


    //Get and save a reference to a Bitmap object
    // containing the contents of the tree file.
    treeBitMap = Bitmap(tree.content);

    //Place the treeBitMap in a new Image object and
    // place it on the canvas near the bottom center of
    // the canvas.
```

```
    treeBitMap.x =
               canvasObj.width / 2 - treeBitMap.width/2;
    treeBitMap.y = canvasObj.height - treeBitMap.height;

    newTreeImage.addChild(treeBitMap);
    this.addChild(newTreeImage);

    //Make the tree almost invisible. It will be made
    // highly visible in conjunction with a
    // lightening flash.
    newTreeImage.alpha = 0.2;

    //Cause the blue background of the tree to
    // be transparent.
    processChromaKey(treeBitMap);


    //Register a timer listener and start the timer
    // running.
    timer.addEventListener(TimerEvent.TIMER, onTimer);
    timer.start();

  } //end creationCompleteHandler
  //-------------------------------------------------//

  //TimerEvent handler. This method is executed each
  // time the timer object fires an event.
  public function onTimer(event:TimerEvent):void {

    //Update the loop counter. Several things depend on
    // this counter.
    loopCntr++;
    if (loopCntr > int.MAX_VALUE-2) {
      //Guard against numeric overflow.
      loopCntr = 0;
    }//end if

    //Play a wind sound every 100th timer event only
    // if a random value is greater than 0.5. This
    // should happen half the time on the average.
    if ((loopCntr % 100 == 0)&& (Math.random() > 0.5)) {
      wind.play();
    }//end if

    //Make random changes to the background color.
    processBackgroundColor();

    //Make changes to the alpha values of the normal
    // and flipped sky images.
    makeTheCloudsMove();
```

```
      //Draw a filled circle on this Canvas object.
      if (!sizzlePlaying) {
        //Erase the circle. Note that this would also
        // erase the lightening bolt if it were done while
        // the sizzle sound is playing.
        graphics.clear();
      }//end if

      //Make the circle move a very small distance to the
      // right. Make it wrap and reappear on the left
      //when it reaches the right side of the window.
      circleX += dx;
      if (circleX > canvasObj.width - radius) {
        circleX = 5 * radius;
      }//end if
      graphics.beginFill(0xffff00);
      graphics.drawCircle(circleX,circleY,radius);
      graphics.endFill();

    }//end onTimer
    //-------------------------------------------------//

    //This function processes the background color. The
    // color changes among various shades of cyan when
    // there is no lightening bolt. The color changes
    // among various shades of dark yellow when there is a
    // lightening bolt.
    private function processBackgroundColor():void {
      if (!sizzlePlaying) {
        //Vary background color when there is no
        // lightening flash.
        if (Math.random() > 0.5) {
          if (greenBkgnd < 250){
            greenBkgnd += 5;
          }//end if
        }else {
          if(greenBkgnd > 5){
            greenBkgnd -= 5;
          }//end if
        }//end else

        if (Math.random() > 0.5) {
          if (blueBkgnd < 250){
            blueBkgnd += 5;
          }//end if
        }else {
          if(blueBkgnd > 5){
            blueBkgnd -= 5;
          }//end if
        }//end else
```

```
    }else {
      //Vary background color during a lightening flash
      if (Math.random() > 0.5) {
        if (greenBkgnd < 245){
          greenBkgnd += 10;
        }//end if
      }else {
        if(greenBkgnd > 10){
          greenBkgnd -= 10;
        }//end if
      }//end else

      if (Math.random() > 0.5) {
        if (redBkgnd < 245){
          redBkgnd += 10;
        }//end if
      }else {
        if(redBkgnd > 10){
          redBkgnd -= 10;
        }//end if
      }//end else
    }//end else

    bkgndColor = (redBkgnd << 16) + (greenBkgnd << 8)
                                            + blueBkgnd;
    setStyle("backgroundColor", bkgndColor);
    setStyle("backgroundAlpha",0.5);

  }//end processBackgroundColor
  //-------------------------------------------------//

  //This function processes the alpha values of the
  // normal and flipped sky images.
  // The change in alpha values of the overlapping
  // images makes it appear that the clouds are
  // moving.
  private function makeTheCloudsMove():void {

    //Change the decreasing or increasing direction of
    // the changes in the alpha value for the normal
    // sky image when the alpha value hits the limits.
    if (normalAlphaDecreasing && (normalAlpha <= 0.1)) {
      normalAlphaDecreasing = false;
    }else if (!normalAlphaDecreasing &&
                           (normalAlpha >= alphaLim)) {
      normalAlphaDecreasing = true;
    }//end if

    //Increase or decrease the alpha value for the
    // normal sky image.
    if (normalAlphaDecreasing) {
```

```
      normalAlpha -= 0.005;
    }else {
      normalAlpha += 0.005;
    }//end else

    //Cause the alpha value for the flipped sky image
    // to go down when the value for the normal sky
    // image goes up, and vice versa.
    flippedAlpha = alphaLim - normalAlpha;

    //Change the alpha values for both sky images.
    normalSky.alpha = normalAlpha;
    flippedSky.alpha = flippedAlpha;
  }//end makeTheCloudsMove
  //-------------------------------------------------//

  //This function creates a flash of lightening. It
  // also makes the tree more visible during
  // the flash and plays a sizzle sound followed by a
  // clap of thunder. This method simply initiates these
  // actions. They are completed later by an event
  // handler registered on the SoundChannel object.
  private function flashLightening():void {

    //Make the tree more visible. Apparently
    // setting the alpha property has no effect on the
    // alpha byte values that have been individually
    // set. Otherwise, the blue background would
    // become visible.
    newTreeImage.alpha = 1.0;


    //Play a sizzle sound to accompany the flash of
    // lightening. Set a flag to prevent another sizzle
    // sound from being played before this one finishes.
    sizzlePlaying = true;
    channel = sizzle.play();
    //Register an event listener that will be called
    // when the sizzle sound finishes playing.
    channel.addEventListener(
            Event.SOUND_COMPLETE, soundCompleteHandler);


    //Change the background color to a dark yellow.
    redBkgnd = 128;
    greenBkgnd = 128;
    blueBkgnd = 0;

  }//end flashLightening
  //-------------------------------------------------//
```

```
    //This method is called each time the sizzle sound
    // finishes playing. Each time it is called, it plays
    // a thunder sound and clears a flag making it
    // possible for another sizzle sound to be played. It
    // also restores the background color to a dark cyan,
    // erases the lightening bolt, and causes the tree to
    // become almost invisible.
    private function soundCompleteHandler(e:Event):void {

      //Allow another sizzle sound to be played now that
      // this one is finished.
      sizzlePlaying = false;
      //Play the thunder immediately following the end of
      // the sizzle sound.
      thunder.play();

      //Switch the background color from dark yellow
      // to the normal background color.
      redBkgnd = 0;
      greenBkgnd = 128;
      blueBkgnd = 128;

      //Erase the lightening bolt. Note that this will
      // also erase the yellow circle.
      canvasObj.graphics.clear();
      //Make the tree almost invisible.
      newTreeImage.alpha = 0.2;

    }//end soundCompleteHandler
    //-------------------------------------------------//

    //This method draws a yellow segmented lightening
    // bolt that is generally random but always ends up
    // at the location where the tree is standing.
    private function drawLightening():void {

      lighteningStartX = Math.floor(Math.random()
                                  * canvasObj.width / 3)
                                  + canvasObj.width / 3;
      lighteningStartY =
                  Math.random() * canvasObj.height / 10;
      lighteningEndX = canvasObj.width / 2 -6;
      lighteningEndY =
                canvasObj.height - treeBitMap.height + 10;

      //Draw a zero width dark yellow line to the starting
      // point of the lightening bolt.
      canvasObj.graphics.lineStyle(0, 0x999900);
      canvasObj.graphics.lineTo(
                     lighteningStartX, lighteningStartY);
```

```
    //Set the line style to a bright yellow line that is
    // four pixels thick.
    canvasObj.graphics.lineStyle(4, 0xFFFF00);

    //Declare working variables.
    var tempX:uint;
    var tempY:uint = lighteningStartY;
    var cnt:uint;

    //Use a for loop to draw a lightening bolt with
    // twenty random segments.
    for (cnt = 0; cnt < 20; cnt++ ) {
      //Compute the coordinates of the end of the next
      // line segment.
      tempX = Math.floor(Math.random()
                                * canvasObj.width / 3)
                                + canvasObj.width / 3;
      tempY = tempY + Math.floor(Math.random()
                          * (lighteningEndY - tempY)/5);
      //Draw the line segment.
      canvasObj.graphics.lineTo(tempX,tempY);
    }//end for loop

    //Draw a line segment to the top of the tree.
    canvasObj.graphics.lineTo(
                        lighteningEndX, lighteningEndY);

    //Make the lightening go to ground.
    canvasObj.graphics.lineTo(
        lighteningEndX,
           lighteningEndY + treeBitMap.height - 20);
  }//end drawLightening
  //-------------------------------------------------//

  //This method is a click handler on the button. It
  // causes the lightening flash to occur and the
  // lightening bolt to be drawn.
  private function onClick(event:MouseEvent):void {
    //Don't create another lightening bolt while the
    // previous one is still in progress.
    if(!sizzlePlaying){
      flashLightening();
      drawLightening();
    }//end if
  }//end onClick
  //-------------------------------------------------//

  //This method identifies all of the pixels in the
  // incoming bitmap with a pure blue color and sets
  // the alpha bytes for those pixels to a value of
  // zero. Apparently those bytes are not affected by
```

```
    // later code that sets the alpha property of the
    // Image object to another value.
    private function processChromaKey(bitmap:Bitmap):void{
      //Get the BitmapData object.
      var bitmapData:BitmapData = bitmap.bitmapData;

      //Get a one-dimensional byte array of pixel data
      // from the bitmapData object. Note that the
      // pixel data format is ARGB.
      var rawBytes:ByteArray = new ByteArray();
      rawBytes = bitmapData.getPixels(new Rectangle(
               0,0,bitmapData.width,bitmapData.height));

      var cnt:uint;
      var red:uint;
      var green:uint;
      var blue:uint;

      for (cnt = 0; cnt < rawBytes.length; cnt += 4) {
        //alpha is in rawBytes[cnt]
        red = rawBytes[cnt + 1];
        green = rawBytes[cnt + 2];
        blue = rawBytes[cnt + 3];

        if ((red == 0) && (green == 0 ) &&
                                          (blue == 255)) {
          //The color is pure blue. Set the value
          // of the alpha byte to zero.
          rawBytes[cnt] = 0;
        }//end if

      }//end for loop
      //Put the modified pixels back into the bitmapData
      // object.
      rawBytes.position = 0;//this is critical
      bitmapData.setPixels(new Rectangle(
                0,0,bitmapData.width,bitmapData.height),
                rawBytes);

    } //end processChromaKey method
    //-------------------------------------------------//
  } //end class
} //end package
```

9 Miscellaneous

This section contains a variety of miscellaneous materials.

NOTE: **Housekeeping material**

- Module name: Combining Sound with Motion and Image Animation
- Files:
  - ActionScript0170\ActionScript0170.htm
  - ActionScript0170\Connexions\ActionScriptXhtml0170.htm

NOTE: **PDF disclaimer:** Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

-end-