Animation Fundamentals*

Richard Baldwin

This work is produced by The Connexions Project and licensed under the
Creative Commons Attribution License †

Abstract

Learn something about how the Flash Player behaves, and learn to use that behavior to write animation projects. Also learn about two different time bases that you can use to control the timing of your ActionScript animation projects.

NOTE: **Click** Animation01 [1] , Animation07 [2] , or Animation01A [3] to run the ActionScript programs from this lesson. *(Click the "Back" button in your browser to return to this page.)*

1 Table of Contents

- Preface (p. 2)
  - General (p. 2)
    - Viewing tip (p. 2)
      - Figures (p. 2)
      - Listings (p. 2)
  - Supplemental material (p. 3)
- General background information (p. 3)
  - A layman's view of the Flash Player (p. 3)
  - Startup considerations (p. 4)
  - Time base considerations (p. 6)
    - Listen for ENTER_FRAME events (p. 6)
    - Listen for TIMER events fired by a Timer object (p. 8)
- Preview (p. 11)
- Discussion and sample code (p. 11)
  - The project named Animation01 (p. 11)
  - The project named Animation07 (p. 14)
  - The project named Animation01A (p. 18)
- Run the projects (p. 19)
- Resources (p. 19)
- Complete project listings (p. 19)
- Miscellaneous (p. 25)

*Version 1.1: May 28, 2010 12:35 pm GMT-5

†http://creativecommons.org/licenses/by/3.0/

[1] http://cnx.org/content/m34485/latest/Animation01.html

[2] http://cnx.org/content/m34485/latest/Animation07.html

[3] http://cnx.org/content/m34485/latest/Animation01A.html

2 Preface

2.1 General

NOTE: All references to ActionScript in this lesson are references to version 3 or later.

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

Several ways to create and launch ActionScript projects

There are several ways to create and launch projects written in the ActionScript programming language. Many, but not all of the lessons in this series will use Adobe Flex as the launch pad for the sample ActionScript projects.

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* [4] *.)* 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 [5] 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

- Figure 1 (p. 7) . Default frame rate for Flash Player 10.
- Figure 2 (p. 8) . Measured frame rate of Flash Player 10 at ten frames per second.
- Figure 3 (p. 9) . Measured Timer event rate at 30 events per second and 30 frames per second.
- Figure 4 (p. 10) . Measured Timer event rate at 10 events per second and 30 frames per second.
- Figure 5 (p. 11) . Measured Timer event rate at 10 events per second and 10 frames per second.

2.2.2 Listings

- Listing 1 (p. 12) . Beginning of the class named Main.
- Listing 2 (p. 12) . Beginning of the constructor.
- Listing 3 (p. 13) . Register an ENTER_FRAME listener.
- Listing 4 (p. 13) . The event handler method named onEnterFrame.
- Listing 5 (p. 14) . Beginning of the Main class for the project named Animation07.
- Listing 6 (p. 15) . Completion of the constructor for the Main class.
- Listing 7 (p. 15) . The ENTER_FRAME event handler.
- Listing 8 (p. 16) . Beginning of the MoveableImage class for the project named Animation07.
- Listing 9 (p. 16) . Embed the image in the swf file.

[4] http://www.dickbaldwin.com/tocFlex.htm
[5] http://www.dickbaldwin.com/tocFlex.htm

- Listing 10 (p. 16) . The constructor for the MoveableImage class. z
- Listing 11 (p. 17) . The moveIt method of the MoveableImage class.
- Listing 12 (p. 18) . New import directives for the project named Animation01A.
- Listing 13 (p. 18) . Instantiation of a new Timer object.
- Listing 14 (p. 18) . Register a TIMER event handler and start the timer.
- Listing 15 (p. 18) . The TIMER event handler.
- Listing 16 (p. 19) . The Main class for the project named TimeBase01.
- Listing 17 (p. 20) . The Main class for the project named TimeBase02.
- Listing 18 (p. 21) . The Main class for the project named Animation01.
- Listing 19 (p. 22) . The Main class for the project named Animation07.
- Listing 20 (p. 23) . The MoveableImage class for the project named Animation07.
- Listing 21 (p. 24) . The Main class for the project named Animation01A.

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 [6] .

3 General background information

I am going to take a very broad view of *computer animation* in this and the next few lessons. A program that causes visual images to *change over time* produces what I am referring to an animation.

Such visual changes can take on many different forms. Perhaps the most common form of animation is the movement *(change in position)* of an object over time. However, animation can involve a change in any visual attribute of an object over time, such as changes in position, color, size, opacity, orientation, etc.

3.1 A layman's view of the Flash Player

Major differences

There are some major differences between writing animation code in ActionScript and writing animation code in Java, C++, and C#.

Those differences generally have to do with the strong tie between the execution of ActionScript code and the Flash Player. *(In some cases, references to the Flash Player in this lesson may also apply to Adobe Air.)*

To understand how to write animation code in ActionScript, you must first understand a little about the Flash Player.

Why is it called a *player?*

There is a good reason that the Flash Player is referred to as a *player* . In many ways, it resembles a DVD player or a VCR player. What I mean by that is that the Flash Player extracts a series of visual images from memory and displays those images sequentially with a *(hopefully)* fixed time interval between the display of one image and the display of the next image in the sequence.

The scene is displayed once during each frame

We can't know exactly what happens at the lowest levels of the software and the display hardware. However, from the viewpoint of the ActionScript programmer, the scene described by the images stored in memory is displayed over and over even if it isn't changing.

> NOTE: According to the *ActionScript 3.0 Bible, 2nd Edition,* by Braunstein, ActionScript 3.0 uses a concept known as *"dirty rectangles"* to avoid physically rendering rectangular groups of pixels that haven't changed.

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

Frames

Each new display of the scene is referred to as a *frame* . As you will see later, it is possible for the ActionScript programmer to simply accept the default rate at which frames are displayed, or to execute code to set the *frame rate* to something other than the default rate.

Not the case in Java, C++, and C#

The inherently strong tie between the images stored in memory and the sequential and repetitive display of those images by the Flash Player does not exist in Java, C++, or C# *(although it can be* ***simulated*** *)* .

> NOTE: There is an open-source programming environment named Processing [7] that uses Java to create an architecture very similar to the Flash Player. The Microsoft **XNA Game Studio** *(C#)* and Game Creators **Dark GDK** *(C++)* also create a similar architecture. Note, however, that these are optional add-ons to the language, which is not the case regarding ActionScript and the Flash Player.

The programmer is responsible for the display

Generally speaking, when writing code in those other languages, it is up to the programmer to write the code that determines how, if, and when the images are displayed on the screen.

While it is not unusual for the programmer to write code to cause images to be displayed in a sequential and repetitive manner in those other languages *(see the above note (p. 4))* , the decision to do that is strictly up to the programmer.

Good news and bad news

> NOTE: It is possible to *(almost)* prevent the Flash Player from displaying repetitive frames by writing code to set the frame rate to 0.01 frames per second. This is not without its problems however.

There are pros and cons to both approaches. The good news is that the behavior of the Flash Player relieves the programmer of the responsibility to display images on a sequential and repetitive basis.

The bad news is that the Flash Player deprives the programmer of the opportunity to make decisions in that regard.

3.2 Startup considerations

What happens when a swf file is loaded?

It is hard to know exactly what happens when a swf file is loaded into the Flash player. I have searched the web extensively and have been unable to find definitive information in that regard.

However, I think I understand what happens, and I will share what I believe to be true with you.

Different ways to write ActionScript 3 programs

You can write ActionScript 3 programs by creating AS3 or ActionScript projects using **Adobe's Flex Builder 3** , **Adobe's Flash Builder 4** , or using the free open source FlashDevelop [8] software.

> NOTE: There are other ways to write ActionScript 3 projects as well.

A skeleton ActionScript source code file

With all three tools, the creation of an ActionScript project causes a skeleton ActionScript source code file with an extension of *.as* to be created. This file must contain a public class definition with a class name that matches the name of the file exclusive of the extension.

The name of the ActionScript source code file

[7] http://processing.org/

[8] http://www.flashdevelop.org/wikidocs/index.php?title=Main_Page

The default name of the source code file differs for the three development tools, but you can rename it if you wish for all three tools. However, you must also be sure to rename the class definition *(and its constructor)* so that the name of the class matches the name of the source code file.

A *release build*

In all three cases, when you create a *release build* for the project, you end up with a folder containing an swf file, an html file, and some other folders and files as a minimum.

The name of the swf file

The name of the swf file also varies among the different development tools and it is best not to change it. It is referenced inside the html file and possibly inside some of the other files as well, so changing the name of the swf file would require you to make corresponding changes in one or more other files.

The name of the html file

The name of the html file also varies among the different development tools but it appears that you can change it with no ill effects.

What happens when the html file is opened in a browser?

Here is what I believe happens when the html file is opened in a browser that has the Flash Player plug-in installed.

> NOTE: This may be entirely wrong from a detailed technical viewpoint, but I will present it for your consideration because it seems to describe what actually happens.

The swf file is opened in the Flash Player

The code inside the html file causes the swf file to be opened in the Flash Player plug-in that has been installed in that browser.

The swf file contains the compiled class definition for the class defined in the source code file mentioned earlier (p. 4) plus a number of other things, including the name of that class. *(You can probably think of a swf file as being similar to a zip file or a Java JAR file; a file that encapsulates other files.)*

An object of the class is instantiated

The Flash Player plug-in extracts the name of the class and calls the constructor for that class to instantiate an object of that class.

The constructor is executed

All of the code contained in the constructor plus all of the code called by code in the constructor, plus all of the code called by that code, etc., is executed.

Objects of type *DisplayObject* are displayed

If any of that code instantiates objects that derive from the **DisplayObject** class and adds those objects to the *display list* , those objects will be displayed during the next frame.

The current contents of the display list are displayed during each frame

If all of that code terminates without doing something to cause the contents of the display list to be modified or to cause the attributes of those display objects to be modified in the future, they will continue to be displayed once during each frame. In that case, the display will appear to be static insofar as the user is concerned.

> NOTE: Note that some objects, such as **Button** objects, inherently know how to modify their own attributes under certain circumstances, such as being rolled over or clicked by the mouse.

Additional code may be executed in the future

The code that is executed and caused to be executed by the constructor may do something to ensure the future execution of additional code, *(such as registering TIMER or ENTER_FRAME event listeners).*

Code that is executed in the future may modify the display list, may modify the attributes of existing objects on the display list, may instantiate new display objects and add them to the display list, etc. Such changes will be reflected in the visual screen display when they occur.

Frame-to-frame changes in the display

As a result of code that is executed in the future, the display list may change on a frame-to-frame basis causing the physical display to also change on a frame-to-frame basis. In that case, the display won't appear to be static insofar as the user is concerned.

That will be the case for the animation projects that I explain in this lesson.

3.3 Time base considerations

In many cases when writing animation code, it is appropriate to use a stable time base to control progress through the program. There are at least two different ways to access a time base when writing ActionScript code:

- Listen for **ENTER_FRAME** events
- Listen for **TIMER** events fired by an object of the **Timer** class

Not independent approaches

Note, however, that these two approaches are not independent of one another. It is easy to write code that uses the real-time clock to show that simply changing the frame rate will cause the **TIMER** event rate to error from its specified value.

Also, unless the method named **updateAfterEvent** is called in the **TIMER** event handler, it does very little good to use a **Timer** object with a fast event rate in an attempt to produce smooth animations if the frame rate is slower than the **Timer** event rate. Without a call to that method in the event handler, changes made to the visual state of the images in accordance with the **TIMER** event rate will only appear on the screen at the slower frame rate.

Wheels turning backwards

When I was a child, I often went to the movie theatre on Saturday afternoon to watch grade-B western movies featuring stars like Hopalong Cassidy, Roy Rogers, Red Ryder, and others.

There was almost always a chase scene in which the bandits were chasing a stage coach. As a child, I could never figure out why it often looked like the wheels *(with spokes)* on the stage coach were turning backwards.

An artifact of sampling theory

Now that I understand sampling theory, I also understand what caused the wheels to turn backwards when the stage coach was moving forward. However, an explanation of the phenomenon is beyond the scope of this lesson. *(That may make a good topic for a future lesson.)*

> NOTE: It wouldn't be too difficult to write an ActionScript project to demonstrate this phenomenon by varying the event rate of a **Timer** object relative to the frame rate of the Flash Player, but I don't have time to do that right now.

An interaction with the frame rate

Suffice it at this point to say that the phenomenon results from an interaction between the frame rate of the movie and the speed of motion of the spokes on the wheel. *(Click here [9] for a demonstration of the phenomenon.)*

Other dependencies

As you will see later, there are other dependencies between the **Timer** event rate and the frame rate that aren't so easy to explain.

3.3.1 Listen for ENTER_FRAME events

Process an event for each new display frame

According to Braunstein *(see Resources (p. 19)), "All display objects broadcast **Event.ENTER_FRAME** events before every frame is drawn, making them ideal timing beacons for animation."*

[9] http://www.michaelbach.de/ot/mot_wagonWheel/index.html

NOTE: Unfortunately, Braunstein also implies at several locations i n his excellent book that the accuracy of the repetition rate for **ENTER_FRAME** events may leave a lot to be desired .

We can define and register a listener object that will cause code to be executed each time such an event is fired. Thus, we can use the sequence of **ENTER_FRAME** events as a time base with which to control the progress of our ActionScript programs.

A simple AS3 project

Listing 16 provides the source code for the **Main** class in a very simple AS3 project *(not a Flex project)* named **TimeBase01** that illustrates the processing of **ENTER_FRAME** events for the purpose of measuring the average elapsed time between such events.

The average frame rate

This project computes and displays the average frame rate for each of five consecutive sets of 200 frames. *(The project also shows how to set the frame rate to something other than the default value.)*

Figure 1 shows the text output produced by a single run of this project in *debug* mode on a Windows Vista system.

Default frame rate for Flash Player 10.

```
    1271890629505
1271890636155 200 30.075187969924812
1271890642822 400 29.99850007499625
1271890649487 600 30.007501875468865
1271890656153 800 30.003000300030006
1271890662823 1000 29.98500749625187
```

Figure 1: Default frame rate for Flash Player 10.

The elapsed time

The values in the leftmost column show the elapsed time in milliseconds since Jan 1, 1970 *(known in programming circles as the epoch)* at the beginning of the run and at every 200th frame event thereafter.

The frame count and the frame rate

The values in the middle column show the frame count.

The values in the rightmost column show the frame rate computed from the amount of time required to display each set of 200 frames.

The default frame rate

This run was made without executing code to set the frame rate to a specific value. The values in the rightmost column in Figure 1 show the default frame rate for Flash Player 10 running on a Windows Vista machine. As you can see, the average frame rate is pretty solid at 30 frames per second.

Change the frame rate

Figure 2 shows the project output obtained by removing the comment markers and enabling the following statement in Listing 16.

stage.frameRate = 10;

The purpose of this statement is to set the frame rate for the Flash Player to ten frames per second.

Measured frame rate of Flash Player 10 at ten frames per second.

```
    1271891012974
1271891033098 200 9.938382031405286
1271891053097 400 10.00050002500125
1271891073097 600 10
1271891093097 800 10
1271891113096 1000 10.00050002500125
```

Figure 2: Measured frame rate of Flash Player 10 at ten frames per second.

The average frame rate

As you can see in Figure 2, the average frame rate in this case is solid at ten frames per second.

A best-case scenario

This is probably the optimum case for measuring the frame rate since the program doesn't do anything else that may have an adverse impact on the ability of the Flash Player to maintain a constant frame rate.

Won't explain the code at this time

I'm not going to explain the code in Listing 16 at this time. You should have no difficulty understanding most of the code in Listing 16. Some of the code in Listing 16 is new to this lesson, but I will explain very similar code later in the lesson in conjunction with other projects.

3.3.2 Listen for TIMER events fired by a Timer object

Constructing a Timer object

The **flash.utils** package contains a class named **Timer** . An object instantiated from this class will fire a **TIMER** event every *(specified value)* milliseconds.

The constructor for the class requires two parameters:

1. The delay until the first event and between successive events in milliseconds as type **Number** .
2. The repeat count as type **int** , which specifies the number of events that will be fired. If this value if 0, the timer will fire events indefinitely.

The reciprocal of the rate

Note that whereas the frequency of frame events (p. 7) is specified as *frames per second* , the repetition parameter for timer events is specified as the time interval between events *(milliseconds per event)* . One is the reciprocal of the other.

You must start the timer

The timer does not start automatically. You must call the **start()** method to start it.

A project to measure the TIMER event rate

Listing 17 provides a simple AS3 project that illustrates the use of **TIMER** events. As before, this project computes and displays the average frequency of **TIMER** events for each of five consecutive sets of 200 events.

Two time bases running

In this case, there are two different time bases running: the frame rate and the timer rate. Unfortunately, it appears that they are not independent of one another.

A rate of 30 TIMER events per second

Figure 3 shows the output when the timer is set to fire 30 events per second and the Flash Player is running at its default rate of approximately 30 frames per second.

Measured Timer event rate at 30 events per second and 30 frames per second.

```
    1271892152172
1271892159395 200 27.689325764917623
1271892166061 400 30.003000300030006
1271892172727 600 30.003000300030006
1271892179393 800 30.003000300030006
1271892186062 1000 29.9895036737142
```

Figure 3: Measured Timer event rate at 30 events per second and 30 frames per second.

Not too bad after a slow start

As you can see in Figure 3, after a somewhat slow start during the first 200 events, the timer event rate was reasonably solid at 30 events per second, on the average, for the next 800 events.

A rate of 10 TIMER events per second

Figure 4 shows the output when the timer was set to fire 10 events per second and the Flash Player was running at its default rate of approximately 30 frames per second.

Measured Timer event rate at 10 events per second and 30 frames per second.

```
    1271892525653
1271892548767 200 8.65276455827637
1271892571894 400 8.64790072209971
1271892594995 600 8.657633868663694
1271892618445 800 8.528784648187633
1271892641413 1000 8.707767328456983
```

Figure 4: Measured Timer event rate at 10 events per second and 30 frames per second.

Not a very accurate event rate

In this case, the event rate was relatively *precise* at an average of about 8.65 events per second but didn't fare well in terms of *accuracy* since the target rate was 10 events per second.

Accuracy versus precision

Accuracy and precision don't mean the same thing. According to Wikipedia [10] ,

> *"In the fields of engineering, industry and statistics, the accuracy of a measurement system is the degree of closeness of measurements of a quantity to its actual (true) value. The precision of a measurement system, also called reproducibility or repeatability, is the degree to which repeated measurements under unchanged conditions show the same results."*

Another result at a rate of 10 TIMER events per second

Figure 5 shows the output when the timer was set to produce 10 events per second and the Flash Player was running at a frame rate of 10 frames per second.

[10] http://en.wikipedia.org/wiki/Accuracy_and_precision

Measured Timer event rate at 10 events per second and 10 frames per second.

```
    1271893113643
1271893134531 200 9.574875526618154
1271893155027 400 9.758001561280249
1271893175626 600 9.709209184911888
1271893195926 800 9.852216748768473
1271893216226 1000 9.852216748768473
```

Figure 5: Measured Timer event rate at 10 events per second and 10 frames per second.

Close, but no cigar

In this case, the event rate was closer to the target of 10 events per second, but was still consistently below the mark.

Even more troublesome is the difference between the results in Figure 4 and the results in Figure 5, which show that the event rate of the timer depends on the frame rate of the Flash Player.

An empty Flash Player window

The Flash Player was running in all five cases discussed above, but it was simply displaying an empty white window. No objects were being displayed in the Flash Player window.

A stand alone Flash Player

It is also worth noting that in all five cases, the standalone Flash Player was running, as opposed to the Flash Player plug-in for a browser. Therefore, these results were not influenced by the behavior of any specific browser.

NOTE: *The two projects discussed above were developed using the FlashDevelop tool. By default, the Test Movie option on the FlashDevelop Project Menu runs the project in a stand alone Flash Player and not in a browser plug in.*

4 Preview

In the remainder of this lesson, I will explain three different animation projects.

Run the ActionScript projects

If you have the Flash Player plug-in *(version 9 or later)* installed in your browser, you can click here (p. 1) to run the projects that I will explain in this lesson.

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.

5 Discussion and sample code

5.1 The project named Animation01

I will explain the remaining projects in this lesson in fragments. A complete listing of the code for the project named **Animation01** is provided in Listing 18 near the end of the lesson.

This is a very simple AS3 animation project. *(Note that it is not a Flex project.)*

I recommend that you run (p. 1) the online version of this project before continuing. *(See Deployment of FlashDevelop projects* (p. 25) *for information on how to deploy a release build from a FlashDevelop project on the Connexions website.)*

The ENTER_FRAME event stream

This project uses the **ENTER_FRAME** event stream as a time base to cause a filled blue circle drawn on a transparent **Sprite** object to move diagonally from left to right across the Flash window. The sprite with the circle moves out of the Flash window at the bottom right.

Beginning of the class named Main

This project was developed using the FlashDevelop tool. By default, the FlashDevelop tool names the required source code file and class definition **Main** . Listing 1 shows the beginning of the **Main** class for the project named **Animation01** .

Listing 1: Beginning of the class named Main.

```
  package {
import flash.display.Sprite;
import flash.events.Event;

public class Main extends Sprite{
  private var sprite:Sprite;
  private var dx:Number = 3;//x-movement distance
  private var dy:Number = 2;//y-movement distance
  private var radius:Number = 24;//radius of circle
```

Not much that is new here

The class named **Main** extends the class named **Sprite** , which is a subclass of the class named **DisplayObject** several levels down the inheritance hierarchy. Therefore, the Flash Player will instantiate an object of this class and add it to the display list.

Otherwise, there is nothing in Listing 1 that should be new to you, so no further explanation of the code in Listing 1 should be required.

Beginning of the constructor

The beginning of the constructor for the **Main** class is shown in Listing 2.

Listing 2: Beginning of the constructor.

```
    public function Main():void {
  sprite = new Sprite();
  //Enable the following statement to cause the
  // sprite background to be visible.
  //sprite.opaqueBackground = true;

  //Draw a filled circle on the Sprite object;
  sprite.graphics.beginFill(0x0000ff, 1.0);
  sprite.graphics.drawCircle(radius,radius,radius);
  sprite.graphics.endFill();

  addChild(sprite);//default location is 0,0
```

Another object of type Sprite

Listing 2 instantiates another object of the **Sprite** class and draws a blue filled circle on that object.

Then it adds that object to the display list as a child of the object of the class **Main** .

Draw a filled circle

According to the documentation, the **beginFill** method that is called on the **Sprite** object in Listing 2:

> *"Specifies a simple one-color fill that subsequent calls to other* ***Graphics*** *methods (such as* ***lineTo()*** *or* ***drawCircle()*** *) use when drawing. The fill remains in effect until you call the* ***beginFill()*** *,* ***beginBitmapFill()*** *, or* ***beginGradientFill()*** *method. Calling the* ***clear()*** *method clears the fill. The fill is not rendered until the* ***endFill()*** *method is called."*

That should be a sufficient explanation of how the code in Listing 2 draws a filled circle on the new **Sprite** object.

Add the new Sprite object to the display list

As mentioned earlier, the last statement in Listing 2 adds the new **Sprite** object to the display list as a child of the object of the **Main** class. **By default** , the new **Sprite** object is added at coordinates 0,0 which is the upper-left corner of the **Main** object.

Register an ENTER_FRAME listener

Listing 3 registers an event handler method named **onEnterFrame** that will be executed each time the Flash Player enters a new display frame.

Listing 3 also signals the end of the constructor.

Listing 3: Register an ENTER_FRAME listener.

```
    addEventListener(Event.ENTER_FRAME, onEnterFrame);
  }//end constructor
```

The event handler method named onEnterFrame

The event handler method named **onEnterFrame** is shown in its entirety in Listing 4.

Listing 4: The event handler method named onEnterFrame.

```
      public function onEnterFrame(event:Event):void {
    sprite.x += dx;
    sprite.y += dy;
  }//end onEnterFrame

 }//end class

}//end package
```

Properties named x and y

The **Sprite** object that was instantiated in Listing 2 has properties named **x** and **y** . The documentation has this to say about the property named **x** .

> *"Indicates the x coordinate of the DisplayObject instance relative to the local coordinates of the parent DisplayObjectContainer."*

The description of the property named **y** is very similar.

What this means in practice

What this means in practice is that the Flash Player uses the values of **x** and **y** properties as coordinates to decide where to draw the **Sprite** object relative to the upper-left corner of its container, which in this case is the object of the **Main** class.

By default, these property values are zero, which explains why the new **Sprite** object is initially drawn (p. 13) at the upper-left corner of the **Main** object.

Modify the x and y property values

Each time the Flash Player fires an **ENTER_FRAME** event, the code in Listing 4 increases the values of the **x** and **y** property values of the **Sprite** object by the amounts assigned to **dx** and **dy** in Listing 1.

Draw the sprite in a new location

The next time the objects in the display list are rendered on the screen, the **Sprite** object containing the filled blue circle will be drawn a little further to the right and a little further down the screen. That is what causes the blue ball to move diagonally from left to right across the Flash Player window when you run (p. 1) the project.

Nothing to stop it

Since there is nothing in the code to reverse the process of incrementing the **x** and **y** property values, the blue ball continues moving until it disappears off the Flash Player window on the lower-right side.

I will do something about that in the next project, which causes an image that is a caricature of me to bounce around inside of a rectangle.

5.2 The project named Animation07

A major upgrade

This project is a major upgrade of the project named **Animation01** in several respects. I recommend that you run (p. 1) this project before continuing.

Draw a rectangle

First, the constructor for the **Main** class draws a 450 x 500 pixel rectangle with a yellow background and border with a thickness of three pixels. *(The constructor begins in Listing 5 and continues into Listing 6.)*

An object of a new class

Then the constructor instantiates an object of a new **MoveableImage** class, passing the dimensions of the rectangle to the constructor for that class and adds that object to the display list.

An event handler

Finally, the constructor for the **Main** class registers an **ENTER_FRAME** event handler, which asks the **MoveableImage** object to move each time it is called. *(See Listing 7.)*

The MoveableImage class

The **MoveableImage** class extends the **Sprite** class and embeds an image in the **Sprite** object when it is instantiated. *(See Listing 8 and Listing 9.)*

The dimensions of a rectangle

The constructor for the **MoveableImage** class *(see Listing 10)* receives the dimensions of a rectangle as incoming parameters and saves those dimensions for later use.

The moveIt method

The **MoveableImage** class defines a method named **moveIt** *(see Listing 11)* .

Each time the **moveIt** method is called, the object, *(including the embedded image)* moves by a prescribed distance in the horizontal and vertical directions.

Bounce off the edges

Whenever the object collides with an edge of the rectangle, it bounces off the edge and starts moving in a different direction.

Beginning of the Main class for the project named Animation07

As before, I will explain this program in fragments. A complete listing of the **Main** class for the program is provided in Listing 19.

The **Main** class begins in the fragment shown in Listing 5.

Listing 5: Beginning of the Main class for the project named Animation07.

```
  package {
import flash.display.Sprite;
import flash.events.Event;

public class Main extends Sprite{
  private var moveableImage:MoveableImage
  private var rectWidth:uint = 450;
  private var rectHeight:uint = 500;

  public function Main() {

    //Draw a black rectangle with a yellow background.
    this.graphics.beginFill(0xFFFF00,1.0);
```

The only things that are new in Listing 5 are the two statements that call the **lineStyle** and **drawRect** methods of the **Graphics** class.

The first of the two statements sets the line style for the rectangle to be three pixels thick and to be black.

The second of the two statements sets the upper left corner of the rectangle to a coordinate position of 0,0 in the parent container and sets the width and height to the values established earlier when the width and height variables were declared and initialized.

Completion of the constructor for the Main class

Listing 6 completes the constructor for the **Main** class.

Listing 6: Completion of the constructor for the Main class.

```
      moveableImage =
              new MoveableImage(rectWidth,rectHeight);
    addChild(moveableImage);//default location is 0,0

    addEventListener(Event.ENTER_FRAME, onEnterFrame);
  }//end constructor
```

A new object of a custom class

Listing 6 begins by instantiating an object of the new custom class named **MoveableImage** , passing the width and height of the rectangle to the constructor for that class.

Then Listing 6 adds the new object to the display list. This will cause it to be rendered in the Flash Player window during the next display frame.

Finally, Listing 6 registers an event handler for **ENTER_FRAME** events.

The ENTER_FRAME event handler

The event handler is shown in Listing 7. Each time the **onEnterFrame** method is called, a message is sent to the **MoveableImage** object asking it to execute its **moveIt** method.

Listing 7: The ENTER_FRAME event handler.

```
  public function onEnterFrame(event:Event):void {
    //Ask the image to move.
    moveableImage.moveIt();
  }//end onEnterFrame
```

```
  }//end class

}}//end package
```

The end of the Main class

Listing 7 also signals the end of the **Main** class.

Beginning of the MoveableImage class

A complete listing of the **MoveableImage** class is provided in Listing 20. The beginning of the **MoveableImage** class is shown in the code fragment in Listing 8.

Listing 8: Beginning of the MoveableImage class for the project named Animation07.

```
    package {
  import flash.display.Sprite;
  import flash.events.Event;
  import flash.display.Bitmap;

  public class MoveableImage extends Sprite{

    private var dx:Number = 4;//x-movement distance
    private var dy:Number = 2;//y-movement distance
    private var rectWidth:uint;
    private var rectHeight:uint;
    private var imageWidth:uint;
    private var imageHeight:uint;
```

There is nothing new in Listing 8.

Embed the image in the swf file

The constructor for the **MoveableImage** class continues in Listing 9. The code in Listing 9 extracts an image from the specified image file and embeds it in the swf file with a reference named **headImage** .

Listing 9: Embed the image in the swf file.

```
        [Embed(source='/baldwin.jpg')]
    private var imgClass:Class;
    private var headImage:Bitmap = new imgClass ();
```

New to this lesson

I'm not going to try to explain how and why it works. I will simply suggest that you memorize the syntax for the next time that you need to do the same thing.

In addition, I will refer you to the following website [11] where you will find an explanation.

The constructor for the MoveableImage class

The constructor for the class is shown in its entirety in Listing 10.

Listing 10: The constructor for the MoveableImage class.

```
    public function MoveableImage(rectWidth:uint,
                                  rectHeight:uint) {
```

[11] http://www.bit-101.com/blog/?p=853

```
      //Save the dimensions of the rectangle.
      this.rectWidth = rectWidth;
      this.rectHeight = rectHeight;

      //Get and save the dimensions of the image.
```

The first two statements in Listing 10 simply save the width and height of the rectangle for later use. That shouldn't be new to you.

The next two statements use the reference to the embedded image *(**headImage**)* to get and save the width and height of the image referred to by the instance variable *(see Listing 9)* named **headImage** .

The last statement in Listing 10 adds that image to the display list as a child of the **Sprite** object.

The moveIt method of the MoveableImage class

The **moveIt** method is shown in its entirety in Listing 11.

Listing 11: The moveIt method of the MoveableImage class.

```
    //Cause the Sprite object to move and bounce off of
    // the edges of the rectangle.
    public function moveIt():void {

      //Test for a collision with the left or right edge
      // of the rectangle.
      if (((this.x + dx) > (rectWidth - imageWidth)
                                    || (this.x < 0))) {
        dx *= -1;//Reverse horizontal direction
      }//end if

      //Test for a collision with the top or the bottom
      // of the rectangle.
      if (((this.y + dy) > (rectHeight - imageHeight)
                                    || (this.y < 0))) {
        dy *= -1;//Reverse vertical direction.
      }//end if

      //Make the move.
      this.x += dx;
      this.y += dy;

    }//end onEnterFrame

  }//end class

}}//end package
```

The **moveIt** method modifies the **x** and **y** property values of the **Sprite** object so that it will be rendered in a different location during the next display frame.

The logic that causes the object to bounce off the edges may take a few minutes for you to unravel. Otherwise, you should have no trouble understanding the code in Listing 11.

The end of the class

Listing 11 signals the end of the **MoveableImage** class.

5.3 The project named Animation01A

The code for this project is shown in its entirety in Listing 21.

This project is essentially the same as **Animation01** except that this project creates a **Timer** object and uses **TIMER** events in place of **ENTER_FRAME** events for timing.

If you run (p. 1) this project, you will see that just like **Animation01** , it causes a filled blue circle that is drawn on a transparent **Sprite** object to move diagonally from left to right across the Flash window. The sprite with the circle moves out of the Flash window at the bottom right.

Because of the similarity of this project to the **Animation01** project, I am only going to discuss the code that is significantly different between the two projects.

New import directives for the project named Animation01A

Listing 12 shows two new import directives that are required to make it possible to instantiate an object of the **Timer** class.

Listing 12: New import directives for the project named Animation01A.

```
import flash.events.TimerEvent;
import flash.utils.Timer;
```

Instantiate a new Timer object

Listing 13 shows the code that instantiates a new **Timer** object. This object will fire a **TIMER** event every 30 milliseconds and will continue to fire **TIMER events** indefinitely.

Listing 13: Instantiation of a new Timer object.

```
private var timer:Timer = new Timer(30);
```

Register a TIMER event handler and start the timer

The code in Listing 14 registers an event handler on the **Timer** object and then starts the timer running.

Listing 14: Register a TIMER event handler and start the timer.

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

The TIMER event handler

The **TIMER** event handler is shown in Listing 15. Except for the method signature, this is essentially the same code that you saw in the **ENTER_FRAME** event handler in Listing 4.

Listing 15: The TIMER event handler.

```
public function onTimer(event:TimerEvent):void {
x += dx;
y += dy;
}//end onTimer
```

That's a wrap

I'm going to let that be it for this lesson, which has concentrated on time bases and animation for ActionScript 3 projects.

6 Run the projects

I encourage you to run (p. 1) these projects from the web. Then copy the code from Listing 16 through Listing 21. Use that code to create your own projects. Compile and run the projects. 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 project listings

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

Listing 16: The Main class for the project named TimeBase01.

```
    /*Project TimeBase01

The purpose of this project is to experiment with the use
of Event.ENTER_FRAME as a time base. The project computes
and displays the average frame rate over five consecutive
sets of 200 frames. It also shows how to set the frame
rate to something other than the default value.

Must be run in debug mode to display the text data.
*******************************************************/
package {
  import flash.display.Sprite;
  import flash.events.Event;

  public class Main extends Sprite{
    private var sprite:Sprite;
    private var date:Date;
    private var countA:uint = 0;
    private var baseTime:Number = 0;
    private var currentTime:Number = 0;

    public function Main():void {
      sprite = new Sprite();
      addEventListener(Event.ENTER_FRAME, onEnterFrame);
//      stage.frameRate = 10;
    }//end constructor
    //-------------------------------------------------//

    //Event handler.
    public function onEnterFrame(event:Event):void {
      currentTime = new Date().time;
      if (countA == 0) {
        baseTime = currentTime;
        trace(baseTime);
```

```
      }//end if

      if ((countA > 0) && (countA < 1001)
                                && (countA % 200 == 0)){
        trace(currentTime + " " + countA + " "
               + 1000 / ((currentTime - baseTime) / 200));
        baseTime = currentTime;
      }//end if
      countA++;
    }//end onEnterFrame
  }//end class

}//end package
```

Listing 17: The Main class for the project named TimeBase02.

```
    /*Project TimeBase02

The purpose of this program is to experiment with the use
of a Timer object as a time base. The program computes
and displays the average tick rate of the Timer object
over five consecutive sets of 100 frames . It also shows
how to set the tick rate to a specific value and how to
specify the number of ticks.
*******************************************************/
package {
  import flash.display.Sprite;
  import flash.events.TimerEvent;
  import flash.utils.Timer;

  public class Main extends Sprite{
    private var sprite:Sprite;
    private var timer:Timer = new Timer(1000/10,1001);
    private var date:Date;
    private var countB:uint = 0;
    private var baseTime:Number = 0;
    private var currentTime:Number = 0;

    public function Main():void {
      sprite = new Sprite();
      timer.addEventListener(TimerEvent.TIMER, onTimer);
      timer.start();

//      stage.frameRate = 10;
    }//end constructor
    //-------------------------------------------------//

    //Event handler
    public function onTimer(event:TimerEvent):void {
```

```
      currentTime = new Date().time;
      if (countB == 0) {
        baseTime = currentTime;
        trace(baseTime);
      }//end if

      if((countB > 0) && (countB < 1001) &&
                                  (countB % 200 == 0)){
        trace(currentTime + " " + countB + " "
               + 1000 / ((currentTime - baseTime) / 200));
        baseTime = currentTime;
      }//end if
      countB++;
    }//end onTimer

    //-------------------------------------------------//

  }//end class

}//end package
```

Listing 18: The Main class for the project named Animation01.

```
    /*Project Animation01
Extremely simple animation project.
This is an AS3 project and not a Flex project.

Uses the ENTER_FRAME event for timing.
Causes a filled blue circle that is drawn on a
transparent Sprite object to move diagonally
from left to right across the Flash window.

The sprite with the circle moves out of the
Flash window at the bottom right.
*******************************************************/
pacpackage {
  import flash.display.Sprite;
  import flash.events.Event;

  public class Main extends Sprite{
    private var sprite:Sprite;
    private var dx:Number = 3;//x-movement distance
    private var dy:Number = 2;//y-movement distance
    private var radius:Number = 24;//radius of circle

    public function Main():void {
      sprite = new Sprite();
      //Enable the following statement to cause the
      // sprite background to be visible.
```

```
      //sprite.opaqueBackground = true;

      //Draw a filled circle on the Sprite object;
      sprite.graphics.beginFill(0x0000ff, 1.0);
      sprite.graphics.drawCircle(radius,radius,radius);
      sprite.graphics.endFill();

      addChild(sprite);//default location is 0,0
      addEventListener(Event.ENTER_FRAME, onEnterFrame);
    }//end constructor
    //-------------------------------------------------//

    //Event handler.
    public function onEnterFrame(event:Event):void {
      sprite.x += dx;
      sprite.y += dy;
    }//end onEnterFrame

  }//end class

}//end package
```

Listing 19: The Main class for the project named Animation07.

```
    /*Project Animation07

Classical bouncing ball project written as an AS3 project.
However, in this case the ball is a Sprite object with
an embedded image of Dick Baldwin's caricature.

Causes the image to bounce around inside of a 450 x 500
rectangle.
*******************************************************/
package {
  import flash.display.Sprite;
  import flash.events.Event;

  public class Main extends Sprite{
    private var moveableImage:MoveableImage
    private var rectWidth:uint = 450;
    private var rectHeight:uint = 500;

    public function Main() {

      //Draw a black rectangle with a yellow background.
      this.graphics.beginFill(0xFFFF00,1.0);
      this.graphics.lineStyle(3, 0x000000);
      this.graphics.drawRect(0, 0, rectWidth, rectHeight);
      this.graphics.endFill();
```

```
      moveableImage =
                  new MoveableImage(rectWidth,rectHeight);
      addChild(moveableImage);//default location is 0,0

      addEventListener(Event.ENTER_FRAME, onEnterFrame);
    }//end constructor
    //-------------------------------------------------//

    //Event handler.
    public function onEnterFrame(event:Event):void {
      //Ask the image to move.
      moveableImage.moveIt();
    }//end onEnterFrame

  }//end class

}//end package
```

Listing 20: The MoveableImage class for the project named Animation07.

```
    /*Class MoveableImage
Constructs a Sprite with an embedded image that can be
moved around inside of a rectangle for which the
dimensions are passed to the constructor.
*******************************************************/
package {
  import flash.display.Sprite;
  import flash.events.Event;
  import flash.display.Bitmap;

  public class MoveableImage extends Sprite{

    private var dx:Number = 4;//x-movement distance
    private var dy:Number = 2;//y-movement distance
    private var rectWidth:uint;
    private var rectHeight:uint;
    private var imageWidth:uint;
    private var imageHeight:uint;


    [Embed(source='/baldwin.jpg')]
    private var imgClass:Class;
    private var headImage:Bitmap = new imgClass ();


    public function MoveableImage(rectWidth:uint,
                                  rectHeight:uint) {
      //Save the dimensions of the rectangle.
```

```
      this.rectWidth = rectWidth;
      this.rectHeight = rectHeight;

      //Get and save the dimensions of the image.
      imageWidth = headImage.width;
      imageHeight = headImage.height;

      //Add the image to the display list.
      addChild(headImage);

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

    //Cause the Sprite object to move and bounce off of
    // the edges of the rectangle.
    public function moveIt():void {

      //Test for a collision with the left or right edge
      // of the rectangle.
      if (((this.x + dx) > (rectWidth - imageWidth)
                                    || (this.x < 0))) {
        dx *= -1;//Reverse horizontal direction
      }//end if

      //Test for a collision with the top or the bottom
      // of the rectangle.
      if (((this.y + dy) > (rectHeight - imageHeight)
                                    || (this.y < 0))) {
        dy *= -1;//Reverse vertical direction.
      }//end if

      //Make the move.
      this.x += dx;
      this.y += dy;

    }//end onEnterFrame

  }//end class

}//end package
```

Listing 21: The Main class for the project named Animation01A.

```
    /*Project Animation01A
Extremely simple animation program.
This is an AS3 project and not a Flex project.

This program is essentially the same as Animation01
except that this program creates a Timer object and
```

```
uses TIMER events in place of ENTER_FRAME events for
timing.

Causes a filled blue circle that is drawn on a
transparent Sprite object to move diagonally
from left to right across the Flash window.

The sprite with the circle moves out of the
Flash window at the bottom right.
*******************************************************/
package {
  import flash.display.Sprite;
```

9 Miscellaneous

This section contains a variety of miscellaneous materials.

NOTE: **Housekeeping material**

- Module name: Animation Fundamentals
- Files:
  - ActionScript0150\ActionScript0150.htm
  - ActionScript0150\Connexions\ActionScriptXhtml0150.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.

NOTE: **Deployment of FlashDevelop projects on the Connexions website:** As of the initial publication of this module, the Connexions website requires that all resources be located in the same folder as the file named **index.cnxml** . However, the output from a *release build* with FlashDevelop places a JavaScript file named **swfobject.js** in a folder named **js** that is a child of the folder that contains all of the other files, including the file named **index.html** . You can flatten this structure for deployment on the Connexions website by moving the file named **swfobject.js** into the folder that contains **index.html** , and then modifying one line of html code in the file named **index.html** to cause it to refer to the file named **swfobject.js** in the same folder as itself.

-end-