Fundamentals of Image Pixel Processing*

Richard Baldwin

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

Abstract

Learn how to write skeleton code for creating a Bitmap object from an image file and how to display the bitmap. Learn two different ways to handle the image file, one of which results in a security issue that must be handled at compile time. Learn how to extract the BitmapData object from the Bitmap object and how to use the getPixels, setPixels, and setPixel32 methods to process the pixels in the bitmap. Learn how to apply a color inversion algorithm to invert the colors in a BitmapData object.

NOTE: **Click** Bitmap05 [1] or Bitmap06 [2] 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)
- Preview (p. 5)
- Discussion and sample code (p. 8)
  - The program named Bitmap05 (p. 8)
    - MXML code for the program named Bitmap05 (p. 8)
    - ActionScript code for the program named Bitmap05 (p. 9)
  - The program named Bitmap06 (p. 20)
    - MXML code for the program named Bitmap06 (p. 20)
    - ActionScript code for the program named Bitmap06 (p. 20)

*Version 1.1: May 22, 2010 3:15 pm -0500
†http://creativecommons.org/licenses/by/3.0/
[1]http://cnx.org/content/m34461/latest/Bitmap05.html
[2]http://cnx.org/content/m34461/latest/Bitmap06.html

- Run the programs (p. 23)
- Resources (p. 23)
- Complete program listings (p. 23)
- Miscellaneous (p. 28)

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 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. *(See Baldwin's Flex programming website* [3] *.)* You should study that lesson 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 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. 6) . Screen output for both programs.
- Figure 2 (p. 7) . Program file structure for the program named Bitmap05.

2.2.2 Listings

- Listing 1 (p. 8) . MXML code for the program named Bitmap05.
- Listing 2 (p. 9) . Beginning of the ActionScript Driver class.
- Listing 3 (p. 10) . Start the process of loading the image file.
- Listing 4 (p. 11) . Beginning of the complete event handler.
- Listing 5 (p. 12) . Encapsulate the Bitmap in an Image object.
- Listing 6 (p. 13) . Clone the original Bitmap to create a duplicate Bitmap.
- Listing 7 (p. 13) . Modify the duplicate Bitmap.
- Listing 8 (p. 13) . Beginning of the modify method.
- Listing 9 (p. 14) . Process pixels using the getPixels and setPixels methods.

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

- Listing 10 (p. 15) . Modify the pixels in the rectangular region.
- Listing 11 (p. 15) . Put the modified pixel data back into the same rectangular region.
- Listing 12 (p. 16) . Process pixels using the setPixel32 method.
- Listing 13 (p. 17) . Put borders on the top and bottom edges.
- Listing 14 (p. 18) . Return to the complete event handler.
- Listing 15 (p. 19) . Beginning of the invert method.
- Listing 16 (p. 19) . Apply the inversion algorithm.
- Listing 17 (p. 19) . Put the modified pixel data back into the BitmapData object.
- Listing 18 (p. 20) . Code that is different in the program named Bitmap06.
- Listing 19 (p. 23) . MXML code for the program named Bitmap05.
- Listing 20 (p. 23) . ActionScript code for the program named Bitmap05.
- Listing 21 (p. 23) . ActionScript code for the program named Bitmap06.

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

In an earlier lesson titled **Bitmap Basics** , I explained the differences between Flex projects and ActionScript projects. I also introduced you to the classes named **Bitmap** and **BitmapData** . Now it's time to put that knowledge to work. In this lesson, I will show you how to:

- Load the contents of an image file into a **BitmapData** object encapsulated in a **Bitmap** object.
- Use the **setPixel32** , **getPixels** , and **setPixels** methods to access and modify the color content of the individual pixels that make up an image.

Before getting into that, however, it will be useful to explain how image information is stored in files and in the computer's memory.

Vector graphics versus bitmap graphics

Graphics programming typically involves two main types of data: bitmaps and vector graphics. This lesson deals with bitmap data only. I will deal with vector graphics in a future lesson.

A rectangular array of pixels

When you take a picture with your digital camera, the scene is converted into a rectangular array containing millions of uniformly spaced colored dots. Those dots or picture elements *(pixels)* are stored on the memory card in your camera until you download them into your computer.

Width, height, and color depth

An image that is stored in this way is defined by the width and height of the array of pixels along with the number of bits that are used to define the color.

Up to a point, the more pixels that the camera produces to represent a given field of view, the better will be the image. Similarly, the more bits that are used to store the color, the better will be the overall quality of the image, particularly in terms of subtle shades of color.

The resolution

The number of pixels per unit area is commonly referred to as resolution. For example, the display monitor that I am currently using displays an array of 1280 x1024 pixels in a rectangular area with a diagonal measurement of 19 inches. (This is not a particularly high resolution monitor.)

The color depth

The number of bits used to represent the color of a pixel is commonly referred to as the color depth. Most modern computers routinely use a color depth of 32 bits. Note, however, that some file formats used

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

for the storage and transmission of bitmap graphics data use fewer than 32 bits for the representation of each pixel in an image.

The RGB or ARGB color model

ActionScript supports a computer color model commonly known as the RGB model or the ARGB model. With this model, the color of each pixel is represented by three numeric color values and an optional transparency value *(alpha)* .

Each of the three color values represents the contribution or strength of a primary color: *red, green, and blue.* The final color of the pixel is a mixture of the primary colors. This is similar to a kindergarten student mixing red, green, and blue clay to produce a color that is different from red, green or blue. *(I don't have a clay analogy for transparency, however.)*

The effect of the transparency value

In some cases, the pixel also contains another value referred to as the *alpha* value *(ARGB)* that represents the transparency of the pixel.

Transparency comes into play when you draw a new image over an existing image. If the alpha value for a pixel is zero, there is no change in the color of the existing pixel because the new pixel is totally transparent. *(Although the effect is commonly referred to as transparency, the numeric value is actually proportional to opacity, which is the inverse of transparency.)*

Total opacity

If the alpha value indicates total opacity ***(often represented as either 1.0 or 255)*** , the color of the existing pixel is completely replaced by the color of the new pixel. *(I will explain the difference between 1.0 and 255 later.)*

Partial opacity

If the alpha value falls between 0 and 1.0 *(0 and 255)* , the colors of the existing pixel and the new pixel are combined to produce a new blended color. The result is as if you are viewing the original scene through colored glass.

An unsigned 32-bit chunk of memory

Typically, a pixel is represented in the computer by an unsigned 32-bit chunk of memory. Each of the three primary colors and the alpha value are represented by an eight-bit unsigned byte. The bytes are concatenated to form the 32-bit chunk of memory.

256 levels

This results in 256 levels of intensity for each of the primary colors along with 256 levels of transparency for the alpha byte. For example, if the red, green, and alpha bytes are equal to 255 and the blue byte is zero, the pixel will be displayed as bright yellow on a typical computer screen.

Bitmap image file formats

Different file formats are commonly used to store and transmit image data. It is usually desirable to reduce the size of the file required to store a given image while maintaining the quality of the image. This often results in a tradeoff between file size and image quality.

Different formats use different compression algorithms to reduce the size of the file. The bitmap image formats supported by Adobe Flash Player and Adobe Air are GIF, JPG, and PNG.

The GIF format

The Graphics Interchange Format *(GIF)* is a format that is often used to store low quality images in very small files. The format can store a maximum of 256 different colors and can designate one of those colors to represent a fully transparent pixel. By comparison, the typical ARGB format can represent more than sixteen million colors with 256 levels of transparency for each pixel.

The GIF format would not be very satisfactory for images produced by your digital camera, but it is fine for many purposes such as screen icons where high color quality is not an important consideration.

The JPEG format

This format, which is often written as JPG, was developed by the Joint Photographic Experts Group *(JPEG)* . This image format uses a lossy compression algorithm to allow 24-bit color depth with a small file size.

Lossy compression means that what comes out of the compressed file is not identical to what went in. The loss in picture quality is often acceptable, however, given that the format allows for different degrees of lossiness which is inversely related to the size of the compressed file. All of the digital cameras that I have owned produce JPEG files as the standard output and some of them allow the user to select the degree of compression and hence the degree of lossiness.

The JPEG format does not support alpha transparency. Therefore, it is not suitable as a file format for transmitting images with alpha data between computers.

The PNG format

Apparently there are some patent issues with the GIF format. The Portable Network Graphics *(PNG)* format was produced as an open-source alternative to the GIF file format.

The PNG format supports at least sixteen million colors and uses lossless compression. The PNG format also supports alpha transparency allowing for up to 256 levels of transparency in a compressed format.

The BitmapData class

The **BitmapData** class in ActionScript 3 supports a 32-bit ARGB color model with more than sixteen million colors and 256 levels of alpha transparency per pixel.

4 Preview

I will explain two different programs in this lesson. One is named **Bitmap05** and the other is named **Bitmap06** . Both programs produce the same output, which is shown in Figure 1.

Screen output for both programs.

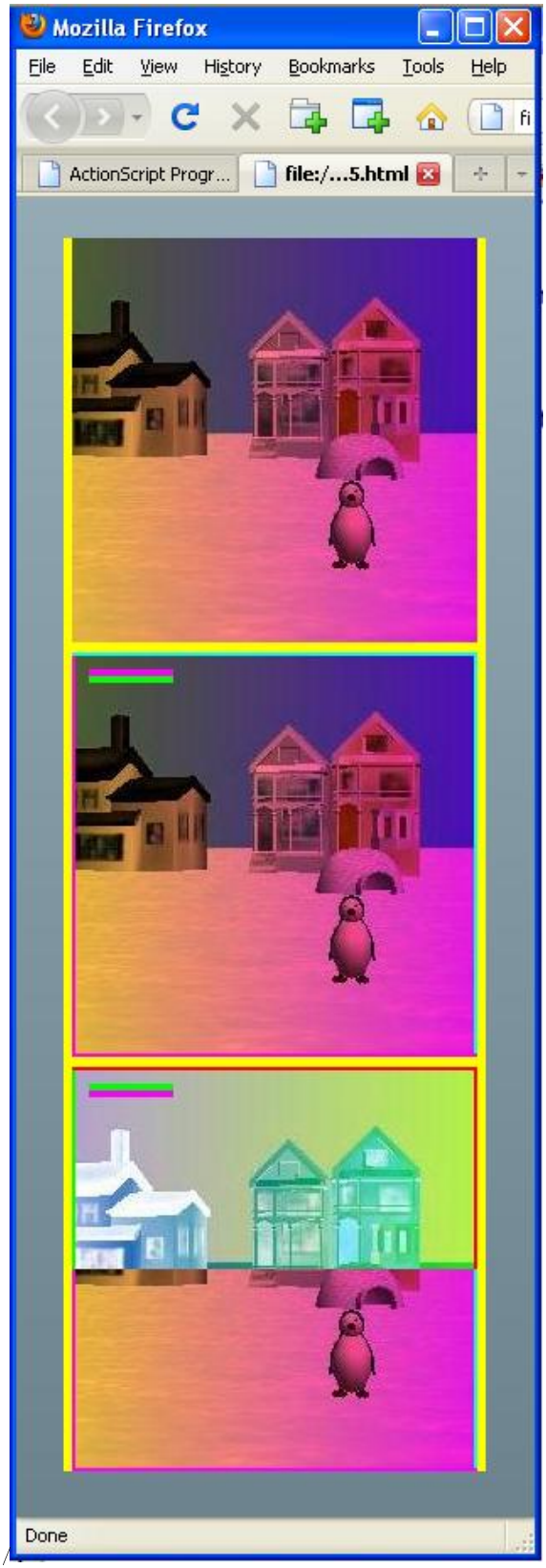

Figure 1: Screen output for both programs.

Program file structure

Figure 2 shows the program file structure taken from the Flex Builder 3 Navigator panel for the program named **Bitmap05** .

Program file structure for the program named Bitmap05.

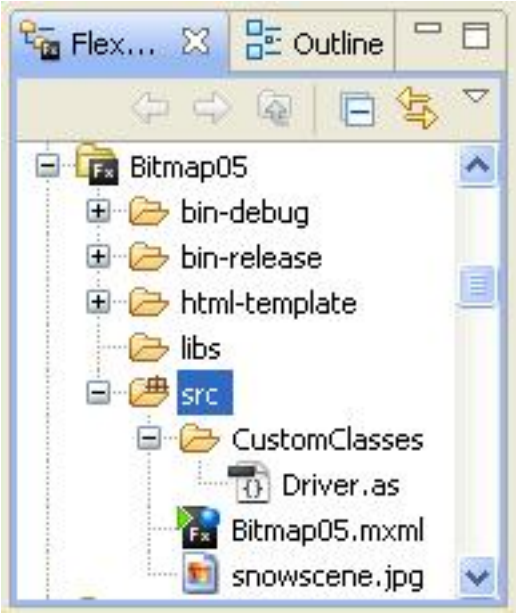

Figure 2: Program file structure for the program named Bitmap05.

The program file structure for **Bitmap06** is the same except for the name of MXML file.

I will use the programs named **Bitmap05** and **Bitmap06** to explain a variety of topics.

Skeleton code

First, I will provide skeleton code for creating a **Bitmap** object from an image file. You will learn how to use the skeleton code to create and display a bitmap from an image file as shown by the top image in Figure 1.

Extract the BitmapData object

I will show you how to extract the **BitmapData** object from the **Bitmap** object so that you can modify the pixels in the image. Once you are able to extract the **BitmapData** object, you will have the bitmap data exposed to the point that you can implement a variety of image processing algorithms such as the following:

- Smoothing
- Sharpening
- Edge detection
- Color filtering
- Color inversion
- Redeye correction

(For more information on how to implement image processing algorithms in general, see the tutorial lessons beginning with Lesson 340 in the section titled ***Multimedia Programming with Java*** *on my web site* [5] *.)*

Using the getPixels, setPixels, and setPixel32 methods

I will explain how to use the **getPixels** , **setPixels** , and **setPixel32** methods for modifying the pixels in a bitmap as shown by the middle image in Figure 1. If you compare the middle image with the top image, you will see that two colored bars were added near the upper-left corner. In addition, colored borders were added to the middle image.

Color inversion

Finally, I will explain how to implement a color inversion algorithm as shown by the bottom image in Figure 1. If you compare the bottom image with the middle image, you will see that the colors of all of the pixels in the top half of the bottom image have been changed. The colors of the pixels were changed using a particular algorithm known widely as a color inversion algorithm.

The color inversion algorithm produces an output that is very similar to an old fashioned color film negative. The algorithm is economical to implement and totally reversible. Therefore, it is used by several major software products to highlight an image and show that the image has been selected for processing.

5 Discussion and sample code

As mentioned earlier, I will explain two different programs in this lesson. One is named **Bitmap05** and the other is named **Bitmap06** .

5.1 The program named Bitmap05

Will explain in fragments

I will explain the code for both programs in fragments. Both programs use the same MXML code but use different ActionScript code. Complete listings of the MXML file and the ActionScript files are provided in Listing 19 through Listing 21 near the end of the lesson.

5.1.1 MXML code for the program named Bitmap05

The MXML code for this program *(and the program named* ***Bitmap06*** *as well)* is shown in Listing 1 and also in Listing 19 for your convenience.

Listing 1: MXML code for the program named Bitmap05.

```
    <?xml version="1.0" encoding="utf-8"?>
<!--
This program illustrates loading an image and
modifying the pixels in the image.
-->

<mx:Application
  xmlns:mx="http://www.adobe.com/2006/mxml"
  xmlns:cc="CustomClasses.*">

  <cc:Driver/>

</mx:Application>
```

[5] http://www.dickbaldwin.com/tocadv.htm

Both programs are written almost entirely in ActionScript. As you can see, the MXML code instantiates a single object of the **Driver** class. Beyond that point, the behavior of both programs is controlled entirely by ActionScript code in the **Driver** class.

5.1.2 ActionScript code for the program named Birtmap05

A security issue

Let me begin by saying that I make no claims of expertise regarding security issues and the Flash Player.

No embedded image

I did not embed the image file shown in Figure 1 in the SWF file for the program named **Bitmap05** . Instead, I included it as a separate file in the *Release Build* of the program. As a result, it was necessary for me to change one of the XML elements in the following configuration file to make it possible for you to download and run (p. 1) the online version of the program.

C:\Program Files\Adobe\Flex Builder 3\sdks\3.2.0\frameworks\flex-config.xml

(Note that the configuration file may be in a different location on your computer.)

The required change

The required change was to set the value in the following element to false instead of true:

```
<use-network>false</use-network>
```

Table 1

Why am I telling you this?

The ActionScript documentation seems to take for granted that you must modify the configuration file to handle the security issue. However, it took a very long time and a lot of searching for me to discover that in order to select certain compiler options, it is necessary to physically modify the configuration file shown above.

I knew generally the kind of change that was required, but I was expecting to find an option in the Flex Builder 3 IDE to allow me to change the compiler options on a project by project basis. If that capability exists in the IDE, I was unable to find it. *(Of course, once you know about the requirement, you can Google "flex-config.xml" and find hundreds of references to the topic.)*

I am telling you this in the hope that this information will save you countless hours of searching through the documentation to discover why you get a runtime error when you replicate this project and then try to download and run it in the Flash Player plugin.

Beginning of the ActionScript Driver class

The MXML code shown in Listing 1 instantiates an object of the **Driver** class. The ActionScript **Driver** class begins in Listing 2.

Listing 2: Beginning of the ActionScript Driver class.

```
 package CustomClasses{
import flash.display.Bitmap;
import flash.display.BitmapData;
import flash.display.Loader;
import flash.events.*;
import flash.geom.Rectangle;
import flash.net.URLRequest;
import flash.utils.ByteArray;
```

```
import mx.containers.VBox;
import mx.controls.Image;
import flash.system.Security;
//=====================================================//

public class Driver extends VBox {
  private var url:String = "snowscene.jpg";//image file

  public function Driver(){//constructor
    //Make the VBox visible.
    setStyle("backgroundColor",0xFFFF00);
    setStyle("backgroundAlpha",1.0);
```

Establish the name and path of the image file

The class definition in Listing 2 begins by declaring and populating an instance variable named **url** with the name of the image file shown in Figure 2. As you can see in Figure 2, this file was located in the same folder as the MXML file. Therefore, no path information was required to specify the location of the image file.

The constructor

The constructor for the class also begins in Listing 2. This class extends the **VBox** class. The code in Listing 2 causes the background color of the **VBox** object to be yellow and also causes the yellow background to be completely opaque. You can see the opaque yellow background of the **VBox** object in Figure 1.

Opacity is often represented as either 1.0 or 255

Remember that I told you earlier (p. 4) that the opacity value is *"often represented as either 1.0 or 255."* Usually when you set the alpha value as a property of a Flex component, you must specify a value ranging from 0.0 to 1.0 with a value of 1.0 being completely opaque. On the other hand, when you are dealing with the actual alpha value in bitmap data, you must specify a value ranging from 0 to 255 with a value of 255 being completely opaque.

Start the process of loading the image file

Listing 3 starts the process of loading the image file. As I mentioned earlier, this program does not embed the image file in the SWF file. *(The program named **Bitmap06** , which I will explain later, does embed the image file in the SWF file.)* Instead, the image file for this program ends up as a separate file on the server that must be downloaded in addition to the SWF file. *(The ActionScript literature contains numerous discussions regarding the pros and cons of embedding versus not embedding resource files.)*

Listing 3: Start the process of loading the image file.

```
    var loader:Loader = new Loader();

  //Register event listeners on the load process
  loader.contentLoaderInfo.addEventListener(
                     Event.COMPLETE,completeHandler);
  loader.contentLoaderInfo.addEventListener(
               IOErrorEvent.IO_ERROR,ioErrorHandler);

  var request:URLRequest = new URLRequest(url);
  loader.load(request);
```

Straightforward code

The code in Listing 3 is fairly straightforward. You should be able to understand it if you analyze it using the ActionScript documentation.

The Event.COMPLETE event handler

The main thing that I want to emphasize from Listing 3 is the registration of the **Event.COMPLETE** event handler. Note that this event handler is registered on the **contentLoaderInfo** property of the **Loader** object and not on the **Loader** object itself. The documentation has this to say about this property:

"Returns a LoaderInfo object corresponding to the object being loaded. LoaderInfo objects are shared between the Loader object and the loaded content object. The LoaderInfo object supplies loading progress information and statistics about the loaded file.

Events related to the load are dispatched by the LoaderInfo object referenced by the contentLoaderInfo property of the Loader object. The contentLoaderInfo property is set to a valid LoaderInfo object, even before the content is loaded, so that you can add event listeners to the object prior to the load."

The Event.COMPLETE event

The documentation has this to say about the **Event.COMPLETE** event:

"Dispatched when data has loaded successfully. In other words, it is dispatched when all the content has been downloaded and the loading has finished. The complete event is always dispatched after the init event. The init event is dispatched when the object is ready to access, though the content may still be downloading."

Beginning of the complete event handler

The **complete** event handler that is registered in Listing 3 begins in Listing 4. This handler is executed when the load process is complete and the image data is available.

Listing 4: Beginning of the complete event handler.

```
  private function completeHandler(event:Event):void{

//Get, cast, and save a reference to a Bitmap object
// containing the content of the image file.
var original:Bitmap = Bitmap(
                      event.target.loader.content);

//Set the width and height of the VBox object based
// on the size of the original bitmap.
this.width = original.width + 10;
this.height = 3*original.height + 12;
```

Get a reference to the Bitmap object

The handler begins by using the incoming reference to the **Event** object to execute a complex statement that ends up with a reference to the **Bitmap** object. However, that reference is received as type **DisplayObject** and must be cast to type **Bitmap** to be used for its intended purpose in this program. The reference is cast to type **Bitmap** and saved in the variable named **original** .

When the first statement in Listing 4 finishes executing, the variable named **original** refers to a **Bitmap** object containing the image from the image file specified in Listing 2.

Set the dimensions of the VBox object

After creating the new **Bitmap** object, Listing 4 uses the dimensions of the **Bitmap** object to set the dimensions of the **VBox** object, which is shown by the yellow background in Figure 1.

Can use almost any image

All of the placement information for the images shown in Figure 1 is based on the dimensions of the **Bitmap** object. Therefore, you should be able to substitute any JPEG, PNG, or GIF image file in place of my file so long as the name of the file matches the name and location of the file specified in Listing 2. Note

however that your image file will need to be wide enough and tall enough to prevent the magenta and green color bars added to the center image in Figure 1 from extending outside the image.

Dealing with a type compatibility issue

In the earlier lesson titled **Bitmap Basics** , I explained that in order to add a child to a **VBox** object, that child:

- Must be a subclass of the **DisplayObject** class and
- Must implement the **IUIComponent** interface.

While a **Bitmap** object is a subclass of **DisplayObject** , it does not implement the **IUIComponent** interface. Therefore, it is not compatible with being added directly to the **VBox** object. I resolved the issue in that lesson by encapsulating the **Bitmap** object in an object of the **UIComponent** class, which implements the **IUIComponent** interface.

Encapsulate the Bitmap in an Image object

In this lesson, I decided to be more specific and encapsulate the **Bitmap** object in an object of the **Image** class. This is allowable because the **Image** class is a subclass of the **UIComponent** class.

Listing 5 encapsulates the **Bitmap** in an **Image** object and adds it to the **VBox** object to be displayed at the top of and five pixels to the right of the left edge of the **VBox** as shown by the top image in Figure 1.

Listing 5: Encapsulate the Bitmap in an Image object.

```
    //Encapsulate the bitmap in an Image object and add
// the Image object to the VBox. Display it at
// x=5 and y=0
original.x = 5;
original.y = 0;
var imageA:Image = new Image();
imageA.addChild(original);
this.addChild(imageA);
```

A curious situation

This brings up a curious situation regarding the placement of the **Image** objects in the **VBox** object. Normally, if you instantiate **Image** objects and populate them directly from the contents of image files ***(by calling the load method on the Image object)*** , you can add them to a **VBox** object without the requirement to specify the locations at which the images will be displayed. The layout management rules of the **VBox** object determine how they are displayed.

This case is different

In this case, however, if you instantiate **Image** objects and populate them with **Bitmap** objects by calling the **addChild** method as shown in Listing 5, you must specify the display locations of the **Bitmap** objects within the **VBox** object. If you don't, they all end up in the upper-left corner of the **VBox** .

Honoring the boundaries of the VBox object

Also, if you specify the dimensions of the **VBox** object and add more images of the first type (p. 12) than the size of the **VBox** object can accommodate, scroll bars automatically appear on the **VBox** object.

In this case, however, if you specify the locations such that the **Image** objects won't all fit within the boundaries of the **VBox** object, the images extend outside the bounds of the **VBox** object.

I will leave it as an exercise for the student to sort through all of that.

Clone the original Bitmap to create a duplicate Bitmap

We have now reached the point where we could access the **BitmapData** object encapsulated in the **Bitmap** object and modify the pixel data that comprises the image. However, instead of modifying the

pixels in the original **Bitmap** , I elected to create a duplicate bitmap and modify the pixels in the duplicate. That makes it possible to compare the unmodified image *(top image in Figure 1)* with the modified image *(middle image in Figure 1)*.

Listing 6 calls the **clone** method on the original **Bitmap** object to create a duplicate **Bitmap** object, and saves the duplicate **Bitmap** object's reference in the variable named **duplicateB** .

Listing 6: Clone the original Bitmap to create a duplicate Bitmap.

```
    var duplicateB:Bitmap = new Bitmap(
                        original.bitmapData.clone());

duplicateB.x = 5;
duplicateB.y = original.height;

var imageB:Image = new Image();
imageB.addChild(duplicateB);
this.addChild(imageB);
```

Display the duplicate bitmap

Then Listing 6 adds the duplicate bitmap to a new **Image** object, positions the duplicate bitmap immediately below the top image in Figure 1 and adds the new image to the **VBox** . This is the middle image in Figure 1.

Another curious circumstance

Curiously, the middle image in Figure 1 is five or six pixels further down than I expected it to be. This produces a gap of five or six pixels between the top two images in Figure 1. I am unable to explain the reason for the gap at this time, but I suspect that it may have something to do with the layout rules of the **VBox** container object. When I place two or more ordinary **Image** objects in a **VBox** container, they appear in a vertical stack separated by about five or six pixels. However, that is total speculation on my part.

Modify the duplicate Bitmap

Listing 7 calls the **modify** method passing a reference to **duplicateB** as a parameter. This causes the middle image in Figure 1 to be modified in two different ways.

- First, the magenta and green rows of pixels are inserted near the upper left corner.
- Then a colored border two pixels thick is inserted around the four edges of the bitmap.

Listing 7: Modify the duplicate Bitmap.

```
    modify(duplicateB);
```

Explain the modify method

At this point, I will put the explanation of the **complete** event handler on hold and explain the method named **modify** .

Beginning of the modify method

The **modify** method begins in Listing 8.

Listing 8: Beginning of the modify method.

```
  private function modify(bitmap:Bitmap):void{

var bitmapData:BitmapData = bitmap.bitmapData;
```

The incoming **Bitmap** object encapsulates a **BitmapData** object, which is referenced by a property of the **Bitmap** object named **bitmapData** . Listing 8 gets a copy of that reference and saves it in a local variable named **bitmapData** .

Process pixels using the getPixels and setPixels methods

Listing 9 begins by instantiating a new empty object of type **ByteArray** .

Listing 9: Process pixels using the getPixels and setPixels methods.

```
    var rawBytes:ByteArray = new ByteArray();
rawBytes = bitmapData.getPixels(
                        new Rectangle(10,10,50,8));
```

The ByteArray class

According to the documentation [6] ,

"The ByteArray class provides methods and properties to optimize reading, writing, and working with binary data."

A **ByteArray** object is an object that can be used to store and access data using either square bracket notation [] or method calls. The main benefit of using this data structure from our viewpoint is that it will decompose the 32-bit integers into 8-bit bytes and allow us access the pixel data one byte at a time. Otherwise it would be necessary for us to perform the decomposition ourselves using bit shift operations.

The getPixels method

Listing 9 populates the **ByteArray** object by calling the **getPixels** method on the **BitmapData** object. According to the documentation [7] , this method

*"Generates a byte array from a **rectangular region** of pixel data. Writes an unsigned integer (a 32-bit unmultiplied pixel value) for each pixel into the byte array."*

A Rectangle object

A new **Rectangle** object is instantiated to define the *rectangular region* from which the pixels are extracted. According to the documentation, the constructor for this class

*"Creates a new **Rectangle** object with the top-left corner specified by the x and y parameters and with the specified width and height parameters."*

The rectangular region specified by the parameters in Listing 9 has its upper-left corner at (10,10) , is 50 pixels wide, and 8 pixels high. This is the rectangular occupied by the magenta and green horizontal bars near the upper-left corner of the middle image in Figure 1.

The ByteArray object is populated with pixel data

When the **getPixels** method returns in Listing 9, the pixels from that rectangular region are stored in the **ByteArray** object referred to by **rawBytes** .

The organization of the pixel data

The first four bytes in the array belong to the pixel in the upper-left corner of the rectangular region. The next four bytes belong to the pixel immediately to the right of that one. The array is populated by the data from the rectangular region on a row by row basis.

Each set of four bytes represent one pixel in ARGB format. In other words, the first byte in the four-byte group is the alpha byte. That byte is followed by the red byte, the green byte, and the blue byte in that order. This information is critical when time comes to use the data in the array to modify the pixel data.

The general procedure

The general procedure when using this approach is to extract a rectangular region of pixels into the array, modify the array data, and then call the **setPixels** method to write the modified color data back into the area of memory that represents the rectangular region from the bitmap data.

A very useful format

[6] http://livedocs.adobe.com/flex/3/langref/flash/utils/ByteArray.html
[7] http://livedocs.adobe.com/flex/3/langref/flash/display/BitmapData.html#getPixels%28%29

The format of the data in the **ByteArray** object is very useful when you need to modify consecutive pixels on a row by row basis. It is less useful, but can be used when you need to modify pixels whose locations are more random in nature.

In our case, we want to set the color of all the pixels in the top four rows of the rectangular region to magenta and we want to set the color of all the pixels in the bottom four rows of the rectangular region to green as shown by the middle image in Figure 1. This data format is ideal for that kind of operation.

Modify the pixels in the rectangular region

Listing 10 sets the colors of the pixels in the top four rows to magenta and sets the colors of the pixels in the bottom four rows to green without modifying the value of the alpha byte.

Listing 10: Modify the pixels in the rectangular region.

```
    var cnt:uint = 1;
while(cnt < rawBytes.length){
  if(cnt < rawBytes.length/2){
    rawBytes[cnt] = 255;
    rawBytes[cnt + 1] = 0;
    rawBytes[cnt + 2] = 255;
  }else{
    rawBytes[cnt] = 0;
    rawBytes[cnt + 1] = 255;
    rawBytes[cnt + 2] = 0;
  } //end if-else

  cnt += 4;//Increment the counter by 4.
}//end while loop
```

The magenta and green color values

A magenta pixel is produced by setting the red and blue color bytes to full strength *(255)* and setting the green color byte to 0. A green pixel is produced by setting the red and blue color bytes to 0 and setting the green color byte to 255.

You should be OK by now

Knowing what you now know, you should have no difficulty understanding how the data in the **ByteArray** object is modified to produce the magenta and green colored areas near the upper-left corner of the middle image of Figure 1.

Not quite finished yet

Note, however that we haven't modified the actual pixel data in the bitmap yet. So far we have made a copy of all the pixel data in the rectangular region and have modified the color values in the copy of the pixel data. We still need to write the modified pixel data back into the **BitmapData** object to actually modify the image.

Put the modified pixel data back into the same rectangular region

Listing 11 calls the **setPixels** method to store the pixel data that is contained in the **rawBytes** array back into the same rectangular region of the bitmap image.

Listing 11: Put the modified pixel data back into the same rectangular region.

```
    rawBytes.position = 0;//this is critical
bitmapData.setPixels(
                new Rectangle(10,10,50,8),rawBytes);
```

The position property of the ByteArray object

With one exception, you should have no difficulty understanding the code in Listing 11. That exception has to do with the **ByteArray** property named **position** . Here is what the documentation has to say about the position property:

"Moves, or returns the current position, in bytes, of the file pointer into the ByteArray object. This is the point at which the next call to a read method starts reading or a write method starts writing."

Whether or not you understand what that means, it is critical that you set the value of the **position** property to zero before calling the **setPixels** method to cause all of the data in the array to be written into the **BitmapData** object. Otherwise, you will get a runtime error.

Some things worth noting

A couple of things are worth noting. First, there was no technical requirement to write the data from the array back into the same rectangular region from which it was read. It could have been written into a different rectangular region in the same bitmap, it could have been written into several different rectangular regions, or it could even have been written into a completely different **BitmapData** object.

No requirement to read the bitmap data

Second, since the code in Listing 10 stored color data into the array that was totally independent of the color values in the **BitmapData** object, there was no requirement to read the color data from the **BitmapData** object in the first place. We could simply have instantiated a new **ByteArray** object and set its length to the product of the width and the height of the rectangular region. Then we could have executed the code in Listing 10 to populate the bytes in the array with magenta and green color values. Then we could have executed the code in Listing 11 to write the pixel data into an appropriate rectangular region in the **BitmapData** object.

On the other hand...

On the other hand, had we wanted to do something like emphasize the green color and deemphasize the blue color in the rectangular region, we would have needed to call the **getPixels** method to get the actual pixel data from the **BitmapData** object into the array. Having that pixel data available, we could have:

- Multiplied the green color value in each pixel by 1.2,
- Multiplied the blue color values in each pixel by 0.8,
- Called the **setPixels** method as in Listing 11 to store the modified pixel data back into the same rectangular region of the **BitmapData** object.

Process pixels using the setPixel32 method

The code in Listing 12 uses the **setPixel32** method to put a magenta border on the left edge of the bitmap and a cyan border on the right edge of the **BitmapData** object as shown in the middle image in Figure 1. *(The border is two pixels thick.)*

Listing 12: Process pixels using the setPixel32 method.

```
    for(var row:uint = 0;row < bitmapData.height;
                                          row++){
  bitmapData.setPixel32(0,row,0xFFFF00FF);
  bitmapData.setPixel32(1,row,0xFFFF00FF);
  bitmapData.setPixel32(bitmapData.width - 1,
                        row,0xFF00FFFF);
  bitmapData.setPixel32(bitmapData.width - 2,
                        row,0xFF00FFFF);
}//end for loop
```

The setPixel32 method and its cousins

The **setPixel32** method and its cousin the **setPixel** method, along with the **getPixel32** method and the **getPixel** method, are completely different from the **getPixels** method and the **setPixels** method used earlier.

And the differences are...

Each call to the **getPixels** method or the **setPixels** method deals with all of the pixels in a specified rectangular region.

Each call to the **getPixel** method, the **getPixel32** method, the **setPixel** method, or the **setPixel32** method deals with only one pixel. That pixel is identified by the horizontal and vertical coordinates of a single pixel in the **BitmapData** object.

Getting and/or setting a pixel value

As you have probably guessed by now, the **getPixel** and **getPixel32** methods are used to return the value of a single pixel from the specified location. Both of these methods return a 32-bit data value of type **uint** .

Similarly, the **setPixel** and **setPixel32** methods are used to write a 32-bit unsigned integer value into a specified location in the **BitmapData** object.

Not decomposed into separate bytes

Unlike with the **getPixels** method used with the **ByteArray** object, the **getPixel** and **getPixel32** methods don't decompose the 32-bit integer value into separate bytes for alpha, red, green, and blue. If you need to separate the returned value into individual bytes, you must accomplish that yourself.

The order of the bytes in the returned value is ARGB. In other words, the leftmost eight bits contain the alpha value, the rightmost eight bits contain the blue value, and the red and green bytes are in the middle.

The difference between the methods

The difference between the methods with 32 in the name the methods without 32 in the name has to do with the alpha byte. The two methods without 32 in the name return a 32-bit unsigned integer but only the 24 RGB bits are meaningful. The eight alpha bits are not meaningful. On the other hand, for the methods with 32 in the name, all four bytes including the alpha byte are meaningful.

Set the pixels to create a border

The code in Listing 12 uses a **for** loop and the **setPixel32** method to set the pixel color to fully opaque magenta *(red plus blue)* for the first two pixels in each row of pixels and to set the pixel color to fully opaque cyan *(green plus blue)* for the last two pixels in each row. This produces a magenta border with a thickness of two pixels on the left edge of the middle image in Figure 1 and produces a cyan border with a thickness of two pixels on the right edge of the middle image in Figure 1.

Put borders on the top and bottom edges

Listing 13 uses similar code to put a cyan border along the top edge and a magenta border along the bottom edge of the middle image in Figure 1.

Listing 13: Put borders on the top and bottom edges.

```
      for(var col:uint = 0;col < bitmapData.width;
                                          col++){
    bitmapData.setPixel32(col,0,0xFF00FFFF);
    bitmapData.setPixel32(col,1,0xFF00FFFF);
    bitmapData.setPixel32(col,bitmapData.height - 1,
                          0xFFFF00FF);
    bitmapData.setPixel32(col,bitmapData.height - 2,
                          0xFFFF00FF);

  } //End for loop

} //end modify method
```

The end of the modify method

Listing 13 also signals the end of the **modify** method. When the method returns from the call that was made in Listing 7, the pixels in the red and green rectangular region near the upper-left corner of the middle image of Figure 1 have been modified relative to the original image shown in the top image in Figure 1. In addition, the pixels along all four edges of the middle image have been replaced by magenta and cyan pixels to produce a border with a thickness of two pixels.

Return to the complete event handler

Returning now to where we left off in the **complete** event handler in Listing 7, the code in Listing 14:

- Creates another duplicate **BitmapData** object.
- Encapsulates it in an **Image** object.
- Places it at the bottom of Figure 1.
- Calls the **modify** method to modify the pixels just like the middle image in Figure 1.
- Calls the **invert** method to invert the colors of all the pixels in the top half of the **BitmapData** object to produce the final image shown at the bottom of Figure 1.

Listing 14: Return to the complete event handler.

```
    //Clone the original bitmap to create another
// duplicate.
var duplicateC:Bitmap = new Bitmap(
                      original.bitmapData.clone());
//Place the duplicateC below the other two in the
// VBox.
duplicateC.x = 5;
duplicateC.y = 2*original.height;

var imageC:Image = new Image();
imageC.addChild(duplicateC);
this.addChild(imageC);

//Modify the pixels as above to add some color to
// the image.
modify(duplicateC);
//Now invert the colors in the top half of this
// bitmap. Note that the magenta and green colors
// swap positions.
invert(duplicateC);

} //end completeHandler
```

Color inversion

The color inversion algorithm is

- Very fast to execute.
- Totally reversible.
- Guaranteed to convert every pixel to a different color.

Usually the new color is readily distinguishable from the old color.

A comparison

If you compare the top half of the bottom image in Figure 1 with the top half of the other two images, you can see the dramatic effect of color inversion. However, all that is required to exactly restore the original colors is to run the inverted color pixels through the inversion process again.

Because of these characteristics, some major software products use color inversion to change the colors in an image that has been selected for processing to provide a visual indication that it has been selected.

The color inversion algorithm

To invert the color of a pixel, you simply subtract the red, green, and blue color values from 255 without modifying the alpha value. To reverse the process, you simply subtract the inverted color values from 255 again, which produces the original color values.

Beginning of the invert method

The **invert** method begins in Listing 15.

Listing 15: Beginning of the invert method.

```
  private function invert(bitmap:Bitmap):void{
//Get the BitmapData object.
var bitmapData:BitmapData = bitmap.bitmapData;

//Get a one-dimensional byte array of pixel data
// from the top half of the bitmapData object
var rawBytes:ByteArray = new ByteArray();
rawBytes = bitmapData.getPixels(new Rectangle(
         0,0,bitmapData.width,bitmapData.height/2));
```

The code in Listing 15 gets the **BitmapData** object encapsulated in the incoming **Bitmap** object and extracts the pixel data from a rectangle that comprises the entire top half of the bitmap data into a **ByteArray** object.

Apply the inversion algorithm

Listing 16 applies the color inversion algorithm to all of the pixel data in the **ByteArray** object by subtracting each color value from 255 and storing the result back into the same element of the **ByteArray** object.

Listing 16: Apply the inversion algorithm.

```
    var cnt:uint = 1;
while(cnt < rawBytes.length){
  rawBytes[cnt] = 255 - rawBytes[cnt];
  rawBytes[cnt + 1] = 255 - rawBytes[cnt + 1];
  rawBytes[cnt + 2] = 255 - rawBytes[cnt + 2];

  cnt += 4;//increment the counter
}//end while loop
```

Put the modified pixel data back into the BitmapData object

Listing 17 calls the **setPixels** method to put the modified pixel data back into the **BitmapData** object producing the final output shown in the bottom image of Figure 1.

Listing 17: Put the modified pixel data back into the BitmapData object.

```
    rawBytes.position = 0;//this is critical
bitmapData.setPixels(new Rectangle(
```

```
                0,0,bitmapData.width,bitmapData.height/2),
                rawBytes);

    } //end invert method
    //-------------------------------------------------//

  } //end class
} //end package
```

This is a case where the new pixel color values depend on the original color values. Therefore, it was necessary to get and use the old color values to compute the new color values.

An interesting side note

If you compare the two color bars in the upper-left corner of the middle and bottom images in Figure 1, you will see that they appear to have swapped positions. This is because the numeric value of magenta is the inverse of the numeric value of cyan and vice versa. The same is true of the cyan and magenta borders.

The end of the program

Listing 17 also signals the end of the program named **Bitmap05** .

5.2 The program named Bitmap06

Behaves just like Bitmap05

This program behaves just like the program named Bitmap05. Therefore, the screen output shown in Figure 1 applies to this program as well as to the program named **Bitmap05** .

The difference between the two programs

This program differs from the program named **Bitmap05** in terms of how and when the image file is loaded. With the program named **Bitmap05** , the image file was maintained as a separate file and downloaded with the SWF file. Then it was loaded at runtime. This resulted in the security issue (p. 9) discussed earlier.

This program embeds the image file into the SWF file at compile time. Since there is no image file to be loaded from the local file system at runtime, the security issue does not apply to this program.

Code modifications

As you might imagine, it was necessary to make some modifications to the program code to accomplish this difference. I will explain those modifications in the following paragraphs.

5.2.1 MXML code for the program named Bitmap06

The MXML code for this program is the same as shown in Listing 19 for the program named **Bitmap05** .

5.2.2 ActionScript code for the program named Bitmap06

I will present and explain only the code that is different from the program named **Bitmap05** . However, a complete listing of the code for this program is provided in Listing 21 near the end of the lesson.

Code that is different in the program named Bitmap06

Listing 18 shows the code that is different in the program named **Bitmap06** . Since some of the code is the same, I have highlighted the code that is different with comments.

Listing 18: Code that is different in the program named Bitmap06.

```
    package CustomClasses{
  import flash.display.Bitmap;
  import flash.display.BitmapData;
  import flash.geom.Rectangle;
```

```
import flash.utils.ByteArray;
import mx.containers.VBox;
import mx.controls.Image;
import mx.events.FlexEvent//different
//=====================================================//

public class Driver extends VBox {
  private var image:Image = new Image();//different

  public function Driver(){//constructor
    //Make the VBox visible.
    setStyle("backgroundColor",0xFFFF00);
    setStyle("backgroundAlpha",1.0);

    [Embed("snowscene.jpg")]//different
    var img:Class;//different

    image.load(img);//different

    //Note that the type of completion event specified
    // here is different from the type of completion
    // event used in Bitmap05.
    //Different
    this.addEventListener(FlexEvent.CREATION_COMPLETE,
                                      completeHandler);
  } //end constructor
  //-----------------------------------------------------//

  //This handler method is executed when the VBox has
  // been fully created. Note that the type of the
  // incoming parameter is more specific than was the
  // case in Bitmap05. However, it isn't used in this
  // program.
  //Different
  private function completeHandler(
                        event:mx.events.FlexEvent):void{
    //Get and save a reference to a Bitmap object
    // containing the content of the image file. This
    // statement is different from Bitmap05.
    //Different
    var original:Bitmap = Bitmap(image.content);

    //Everything beyond this point is identical to
    // Bitmap05 except that the IO error handler was
    // removed. It isn't needed for an embedded image
    // file.
```

A new import directive

The differences begin in Listing 18 with an import directive for the class named **FlexEvent** . This class was not needed in the program named Bitmap05.

Instantiation of an Image object

The differences continue in Listing 18 with the declaration and instantiation of an object of the class **Image** . The embedded image file will be loaded into this object at runtime. Then the **Bitmap** object encapsulated in the **Image** object will be extracted and passed to the **modify** method and the **invert** method the same as before.

Embedding the image file

The two lines of code beginning with the word **[Embed** provide the mechanism by which the image file is embedded into the SWF file. The first line specifies the name and path to the image file. In this case, it was in the same folder as the MXML file so no path was required.

The strange syntax of the **Embed** code means that it really isn't an executable programming statement. Instead, it is an instruction to the compiler telling the compiler to embed the file in the SWF file.

Declare a variable to refer to the embedded file

The line immediately following the **Embed** directive declares a variable of type **Class** named **img** that can later be used to refer to the embedded file.

Load the file contents into the Image object

The second line of code following the **Embed** directive causes the contents of the embedded image file to be loaded into the **Image** object at runtime. Note that the embedded image file is referenced by the variable named **img** that was declared along with the **Embed** directive and passed as a parameter to the load method.

No need to worry about IO errors at runtime

Because the image file is read at compile time and embedded into the SWF file, there is no need to provide an IO error handler that will be executed as a result of a runtime IO error involving the image file. If there is a problem reading the file, that problem will occur when the program is compiled and the SWF file is written.

Register a creationComplete event listener on the VBox

The last statement in the constructor registers a **creationComplete** event handler on the **VBox** . This is considerably different from the program named **Bitmap05** . First, the event handler is registered on the **VBox** instead of being registered on **loader.contentLoaderInfo** . Second, the type of the completion event is different between the two programs.

The **creationComplete** event will be

*"Dispatched when the component (**VBox**) has finished its construction, property processing, measuring, layout, and drawing."*

The assumption is that by the time the event is dispatched, the bitmap data will have been successfully loaded into the **Image** object.

Differences in the creationComplete event handler

The first difference in the complete event handler is the type of event passed to the handler. The type **FlexEvent** shown in Listing 18 is more specialized than the type **Event** shown in Listing 4. However it doesn't matter in this case because the incoming reference to the event object isn't used.

Getting the Bitmap object

The **Bitmap** object in an **Image** object is referred to by the property named **content** .

The last statement that is marked as being different gets a reference to the **Bitmap** object and stores it in the variable named **original** just like was done in Listing 4.

As before, referencing the **content** property returns the **Bitmap** object as type **DisplayObject** . Therefore, it must be cast to type **Bitmap** before it can be used for the intended purpose of this program.

Beyond this point - no changes

Beyond this point, the two programs are identical except that the IO error handler was omitted from this program. As I explained earlier, because the image file is embedded in the SWF file at compile time, there is no need to worry about IO errors involving the image file at runtime.

6 Run the programs

I encourage you to run (p. 1) the online versions of the two programs from the web. Then copy the code from Listing 19 through Listing 21. Use that code to create Flex 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 program listings

Complete listings of the MXML code and the ActionScript code for the programs discussed in this lesson are provided below.

Listing 19: MXML code for the program named Bitmap05.

```
    <?xml version="1.0" encoding="utf-8"?>
<!--
This program illustrates loading an image and
modifying the pixels in the image.
-->

<mx:Application
  xmlns:mx="http://www.adobe.com/2006/mxml"
  xmlns:cc="CustomClasses.*">

  <cc:Driver/>

</mx:Application>
```

Listing 20: ActionScript code for the program named Bitmap05.

```
    /*Bitmap05
Provides skeleton code for creating a Bitmap object from
an image file.
Explains the use of the getPixels, setPixels, and
```

Listing 21: ActionScript code for the program named Bitmap06.

```
    /*Bitmap06
This is an update to Bitmap05 that uses an image that is
embedded in the SWF file rather than a separate
downloaded image file. This eliminates the requirement
to make the following change to the configuration file
at:
```

```
C:\Program Files\Adobe\Flex Builder 3\sdks\3.2.0\
frameworks\flex-config.xml

   <!-- Prevents SWFs from accessing the network. -->
   <use-network>false</use-network>

The behavior of this program is identical to the
behavior of Bitmap05.
*******************************************************/
package CustomClasses{
  import flash.display.Bitmap;
  import flash.display.BitmapData;
  import flash.geom.Rectangle;
  import flash.utils.ByteArray;
  import mx.containers.VBox;
  import mx.controls.Image;
  import mx.events.FlexEvent;
  //=====================================================//

  public class Driver extends VBox {
    private var image:Image = new Image();

    public function Driver(){//constructor
      //Make the VBox visible.
      setStyle("backgroundColor",0xFFFF00);
      setStyle("backgroundAlpha",1.0);

      [Embed("snowscene.jpg")]
      var img:Class;
      image.load(img);

      //Note that the type of completion event specified
      // here is different from the type of completion
      // event used in Bitmap05.
      this.addEventListener(FlexEvent.CREATION_COMPLETE,
                                        completeHandler);
    } //end constructor
    //-----------------------------------------------------//

    //This handler method is executed when the VBox has
    // been fully created. Note that the type of the
    // incoming parameter is more specific than was the
    // case in Bitmap05. However, it isn't used in this
    // program.
    private function completeHandler(
                          event:mx.events.FlexEvent):void{
      //Get and save a reference to a Bitmap object
      // containing the content of the image file. This
      // statement is different from Bitmap05.
      var original:Bitmap = Bitmap(image.content);
```

```
//Everything beyond this point is identical to
// Bitmap05 except that the IO error handler was
// removed. It isn't needed for an embedded image
// file.

//Set the width and height of the VBox object based
// on the size of the original bitmap.
this.width = original.width + 10;
this.height = 3*original.height + 12;

//Encapsulate the bitmap in an Image object and add
// the Image object to the VBox. Display it at
// x=5 and y=0
original.x = 5;
original.y = 0;
var imageA:Image = new Image();
imageA.addChild(original);
this.addChild(imageA);

//Clone the original bitmap to create a duplicate.
var duplicateB:Bitmap = new Bitmap(
                       original.bitmapData.clone());
//Place the duplicate bitmap below the original in
// the VBox. There is a six-pixel downward shift
// that I am unable to explain at this time. The
// shift produces a gap of about six pixels between
// the two images.
duplicateB.x = 5;
duplicateB.y = original.height;

var imageB:Image = new Image();
imageB.addChild(duplicateB);
this.addChild(imageB);

//Modify this duplicate.
modify(duplicateB);

//Clone the original bitmap to create another
// duplicate.
var duplicateC:Bitmap = new Bitmap(
                       original.bitmapData.clone());
//Place the duplicateC below the other two in the
// VBox.
duplicateC.x = 5;
duplicateC.y = 2*original.height;

var imageC:Image = new Image();
imageC.addChild(duplicateC);
this.addChild(imageC);

//Modify the pixels as above to add some color to
```

```
    // the image.
    modify(duplicateC);
    //Now invert the colors in the top half of this
    // bitmap. Note that the magenta and green colors
    // swap positions.
    invert(duplicateC);

  } //end completeHandler
  //-------------------------------------------------//

  //This method modifies the pixels in the incoming
  // bitmap in a variety of ways.
  private function modify(bitmap:Bitmap):void{
    //Get the BitmapData object from the incoming
    // Bitmap object.
    var bitmapData:BitmapData = bitmap.bitmapData;

    //Process pixels using the getPixels and
    // setPixels methods.

    //Get a rectangular array of pixels comprising
    // 50 columns by 8 rows in a one-dimensional
    // array of bytes. The bytes are ordered in the
    // array as row 0, row 1, etc.  Each pixel is
    // represented by four consecutive bytes in ARGB
    // order.
    var rawBytes:ByteArray = new ByteArray();
    rawBytes = bitmapData.getPixels(
                            new Rectangle(10,10,50,8));

    //Set the colors of the top four rows to magenta
    // and the color of the bottom four rows to
    // green. Don't modify alpha.
    var cnt:uint = 1;
    while(cnt < rawBytes.length){
      if(cnt < rawBytes.length/2){
        rawBytes[cnt] = 255;
        rawBytes[cnt + 1] = 0;
        rawBytes[cnt + 2] = 255;
      }else{
        rawBytes[cnt] = 0;
        rawBytes[cnt + 1] = 255;
        rawBytes[cnt + 2] = 0;
      } //end if-else

      cnt += 4;//Increment the counter by 4.
    }//end while loop

    //Put the modified pixels back in the bitmapData
    // object.
    rawBytes.position = 0;//this is critical
```

```
      bitmapData.setPixels(
                     new Rectangle(10,10,50,8),rawBytes);


      //Process pixels using the setPixel32 method.

      //Put a magenta border on the left edge and a
      // cyan border on the right edge. Note that the
      // byte values in the 32-bit pixel are in ARGB order
      // and the border thickness is two pixels.
      for(var row:uint = 0;row < bitmapData.height;
                                                 row++){
        bitmapData.setPixel32(0,row,0xFFFF00FF);
        bitmapData.setPixel32(1,row,0xFFFF00FF);
        bitmapData.setPixel32(bitmapData.width - 1,
                              row,0xFF00FFFF);
        bitmapData.setPixel32(bitmapData.width - 2,
                              row,0xFF00FFFF);
      }//end for loop

      //Put a cyan border along the top edge and a
      // magenta border along the bottom edge.
      for(var col:uint = 0;col < bitmapData.width;
                                            col++){
        bitmapData.setPixel32(col,0,0xFF00FFFF);
        bitmapData.setPixel32(col,1,0xFF00FFFF);
        bitmapData.setPixel32(col,bitmapData.height - 1,
                              0xFFFF00FF);
        bitmapData.setPixel32(col,bitmapData.height - 2,
                              0xFFFF00FF);

      } //End for loop

    } //end modify method
    //-----------------------------------------------------//

    //This method inverts all of the pixels in the top
    // half of the incoming bitmap.
    private function invert(bitmap:Bitmap):void{
      //Get the BitmapData object.
      var bitmapData:BitmapData = bitmap.bitmapData;

      //Get a one-dimensional byte array of pixel data
      // from the top half of the bitmapData object
      var rawBytes:ByteArray = new ByteArray();
      rawBytes = bitmapData.getPixels(new Rectangle(
               0,0,bitmapData.width,bitmapData.height/2));

      //Invert the colors by subtracting each color
      // component value from 255.
      var cnt:uint = 1;
```

```
      while(cnt < rawBytes.length){
        rawBytes[cnt] = 255 - rawBytes[cnt];
        rawBytes[cnt + 1] = 255 - rawBytes[cnt + 1];
        rawBytes[cnt + 2] = 255 - rawBytes[cnt + 2];

        cnt += 4;//increment the counter
      }//end while loop

      //Put the modified pixels back in the bitmapData
      // object.
      rawBytes.position = 0;//this is critical
      bitmapData.setPixels(new Rectangle(
                0,0,bitmapData.width,bitmapData.height/2),
                rawBytes);

    } //end invert method
    //--------------------------------------------------//

  } //end class
} //end package
```

9 Miscellaneous

This section contains a variety of miscellaneous materials.

NOTE: **Housekeeping material**

- Module name: Fundamentals of Image Pixel Processing
- Files:
  - ActionScript0132\ActionScript0132.htm
  - ActionScript0132\Connexions\ActionScriptXhtml0132.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-