Processing Image Pixels using Java, Getting Started

Baldwin shows you how to modify an image by modifying the pixels belonging to that image. He also provides a driver program that makes it easy to modify the pixels in an image and to display the modified image.

Published: September 7, 2004
By Richard G. Baldwin

Java Programming, Notes # 400

Preface

First in a series

This lesson is the first lesson in a new series intended to teach you how to use Java to create special effects with images by directly manipulating the pixels in the images.

Not a lesson on JAI

If you arrived at this lesson while searching for instructions on how to use the Java Advanced Imaging *(JAI)* API, you are certainly welcome to be here. However, that is not the purpose of the lessons in this series. *(Maybe I will write a series on that topic later.)* The purpose of this series is to get right down in the mud and to learn how to implement many of the algorithms that are commonly used to create special effects with images by working directly with the pixels.

Manipulate pixels directly and individually

In this lesson, I will provide and explain a program that makes it easy to:

- Manipulate and modify the pixels that belong to an image
- Display the modified image along with the original image for easy comparison in a *before* and *after* sense

A framework or driver program

This program is designed to be used as a framework or driver that controls the execution of another program that actually processes the pixels.

By using this program as a driver, you can concentrate on writing and executing image-processing algorithms without having to worry about many of the details involving image files, image display, etc.

A simple image-processing program

Also in this lesson, I will provide and explain the first of several image-processing programs designed to teach you how to modify an image by directly modifying the pixels that represent the image.

The image-processing program provided in this lesson will be relatively simple with the intent being to get you started but not necessarily to produce a modified image that is especially interesting.

More interesting imaging processing programs

Future lessons will show you how to write image-processing programs that implement many common special effects as well as a few that aren't so common. This will include programs to do the following:

- Highlight a particular area in an image.
- Blur all or part of an image.
- Sharpen all or part of an image.
- Perform edge detection on an image.
- Apply color filtering to an image.
- Apply color inversion to an image.
- Morphing one image into another image.
- Rotating an image.
- Squeezing part of an image into a smaller size.
- Controlling the brightness of an image using linear and non-linear algorithms.
- Other special effects that I may dream up or discover while doing the background research for the lessons in this series.

Figures 1 through 4 show examples of the first four special effects in the above list.

Viewing tip

You may find it useful to open another copy of this lesson in a separate browser window. That will make it easier for you to scroll back and forth among the different figures and listings while you are reading about them.

Highlighting an image

The special effect illustrated by Figure 1 begins with a picture of a starfish taken in a well-lighted aquarium and converts it to what looks like a picture taken by a SCUBA diver deep underwater.

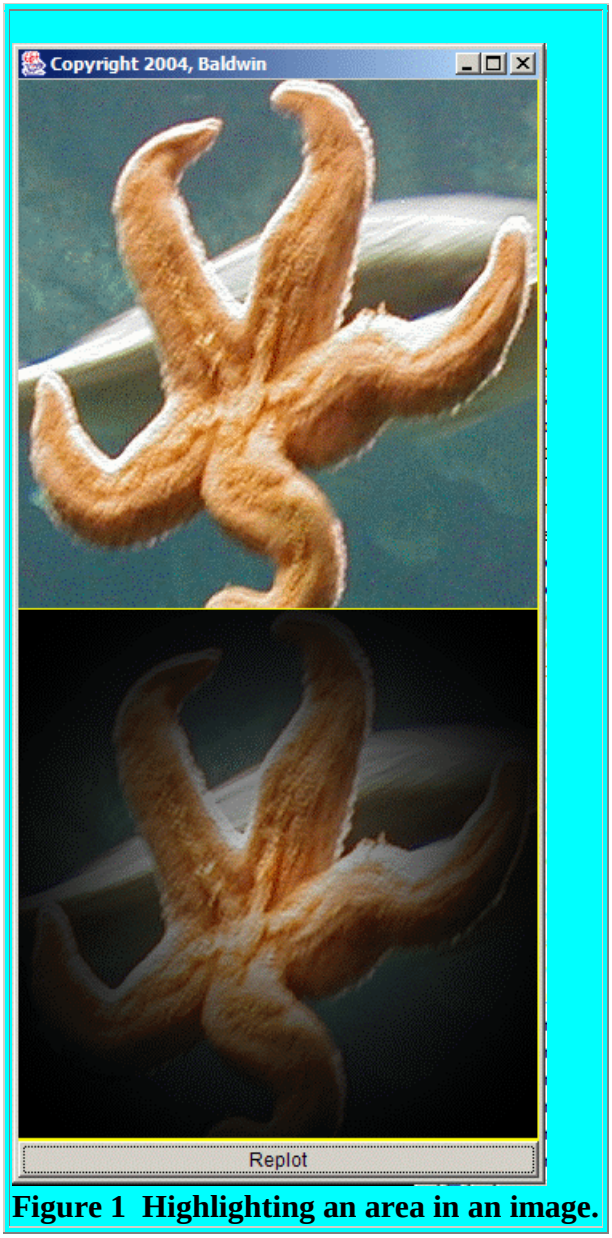

Figure 1 Highlighting an area in an image.

In Figure 1, as in all of the graphic output produced by this program, the original image is shown at the top and the modified image is shown at the bottom.

The program that produced the modified image in Figure 1 allows the user to interactively control the degree to which the light is concentrated in the center of the picture. In other words,

the illumination can range from being concentrated in a very small area in the center to being spread throughout the image.

Blurring an image

Figure 2 illustrates a well-known algorithm that implements the common special effect of blurring an image.

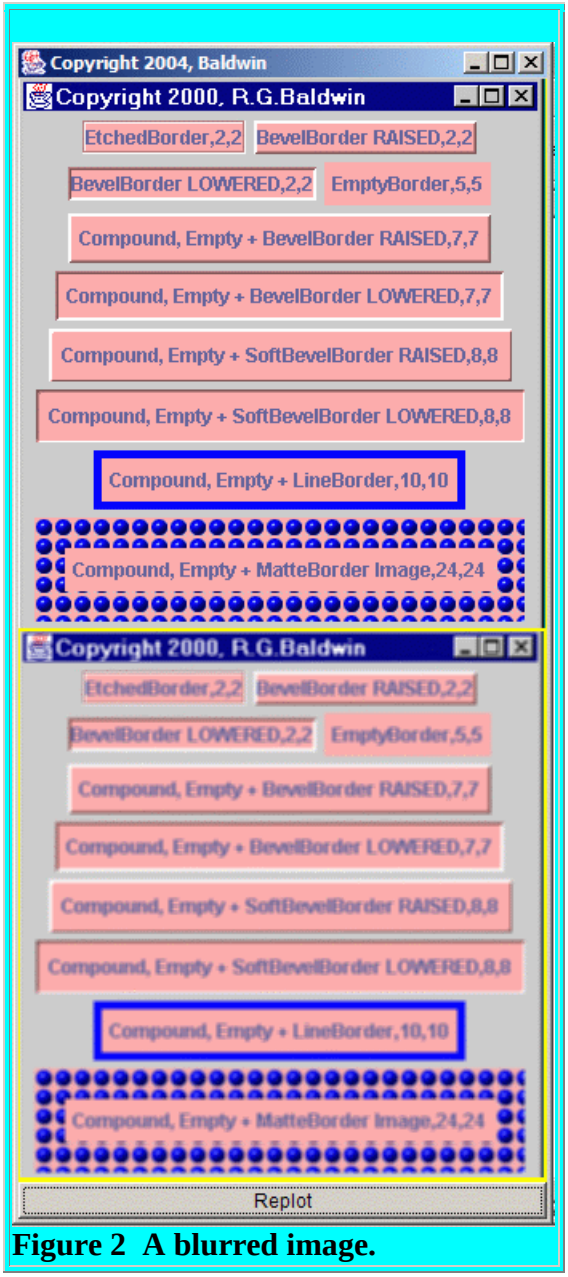

Figure 2 A blurred image.

The original image is at the top, and the blurred image is at the bottom.

The program used to produce Figure 2 allows the user to interactively control the extent of the blurring, ranging from no blurring at all, to extreme blurring.

Sharpening an image

Figure 3 shows the result of applying an algorithm intended to sharpen an image. In general, sharpening is intended to cause the small details in the image to become more visible.

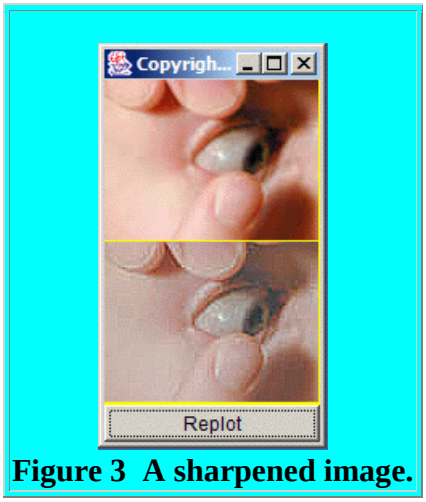

Figure 3 A sharpened image.

The program that produced Figure 3 applies a well-known sharpening procedure. In the final analysis, the success or failure of the algorithm lies in the eye of the beholder, so you will have to make up your own mind as to the results shown in Figure 3.

It does appear to me, however, that some of the fine detail, such as the veins in the eyeball and the ragged edges on the fingernails are enhanced in the processed image at the bottom of Figure 3.

Edge detection

The purpose of edge detection is to highlight the edges of different objects in an image where color changes and shadows produce rapid changes in the color and/or intensity of the image. Other circumstances, such as the edges of letters on a sign will also trigger edge detection on the basis of strong color contrasts between the letters and the background.

Figure 4 shows the result of applying edge detection to a photograph of a person inserting a contact lens into their eye.

Figure 4 An edge-detected image.

You will probably agree that the edge-detection algorithm worked pretty well in this case. It is easy to spot the edges of the fingers, the fingernails, the eyelid, the iris in the eye, and the shadows on the nose.

Theoretical basis and practical implementation

In future lessons, I will provide some of the theoretical basis for special-effects algorithms including those used in Figures 1 through 3.

In addition, I will show you how to implement those algorithms in Java.

In some cases I will mention situations in which the special effect might be useful.

> *(For example, blurring can be used to soften a photograph and can make a person look a few years younger than they actually are by hiding some of the wrinkles. Edge detection can be used to highlight the edges of items in surveillance photos.)*

A disclaimer

The programs that I will provide and explain in this series of lessons are not intended to be used for high-volume production work. Numerous integrated image-processing programs are available for that purpose. In addition, JAI has a number of special effects built in if you prefer to write your own production image-processing programs using Java.

The programs that I will provide in this series of lessons are intended to make it possible for you to develop and to experiment with such algorithms and to gain a better understanding of how they work, and why they do what they do.

Background Information

Image construction, storage, and rendering

Before getting into the programming details, it will be useful to review the concept of how images are constructed, stored, transported, and rendered in Java *(and in most modern computers for that matter).*

An array of colored dots - pixels

A modern computer image, at the point where it is presented *(rendered)* for human consumption, usually consists of a rectangular array of closely spaced colored dots. Ideally, the dots are so small and so close together that the human eye cannot distinguish them individually. This causes them to run together and appear to represent continuous color.

The individual dots are commonly referred to as *pixels*, which I believe is derived from the term *picture elements.*

Image files

The pixels are typically stored and transported in files, and are then extracted from the files and displayed on a computer screen or sheet of paper for human consumption.

There are a fairly large number of formats for storing the pixels in a file. Different file formats have advantages and disadvantages in terms of compression, size, reproduction quality, etc.

Not interested in file formats

This series of lessons will not be concerned about file formats. We will be concerned with what to do with the pixels once they have been extracted from the file. The driver program that I will provide can read **gif** files and **jpg** files, and possibly other file formats as well.

Will pick up at the extraction point for raw pixels

We will initially become interested in the pixels at the point where they have been extracted from the file and exist in the form of a one-dimensional array of type **int**. We will convert that array into a three-dimensional array that is better suited for processing. Once we understand the conversion process, our attention will shift to the three-dimensional array containing pixel data. The image-processing programs that we will write will receive raw pixel data in the form of a three-dimensional array.

A grid of colored pixels

Each three-dimensional array object will represent one image consisting of a grid of colored pixels. When rendered, the pixels in the grid will be arranged in rows and columns. One of the dimensions of the array will represent rows. A second dimension will represent columns. The third dimension will represent the color *(and transparency)* of the pixel.

Now back to the fundamentals

A pixel in a modern computer image is represented by four ***unsigned*** 8-bit bytes of data. Three of those four bytes represent the colors *red*, *green*, and *blue*. The fourth byte, often referred to as the *alpha* byte, represents *transparency*. I will have more to say about the alpha byte and transparency later.

Mixing the primary colors red, green, and blue

Specific colors are created by mixing different amount of red, green, and blue. That is to say, when the program needs to cause the color orange to be displayed on the screen, it mixes together the correct amounts of red, green, and blue to produce orange.

The amounts of each of the three primary colors that are added together to control the overall color of an individual pixel are specified by the individual values stored in the three color bytes for the pixel.

The range of a color

Each unsigned eight-bit color byte can contain 256 different values ranging from 0 to 255 inclusive. If the value of the red byte is 0, for example, no red color is added into the mix to produce the overall color for that pixel. If the value of the red byte is 255, the maximum possible amount of red is added into the mix to produce the overall color for that pixel. The same is true for blue and green as well.

Black and white pixels

If all three of the color pixels have a value of 0, the color of that pixel is black. If all three of the color pixels have a value of 255, the color of that pixel is white. If all three of the pixels have the same value somewhere between 0 and 255, the color of the pixel is some shade of gray.

Sixteen million possible colors

Between black at one extreme and white at the other, there are about sixteen million possible combinations of the three color values, each having 256 possible values. Thus, in theory, the system can produce about sixteen million different colors.

> *(In actuality, it is not likely that there are any monitors, printers, or human eyeballs that can reliably distinguish between sixteen million different colors. For practical purposes, many of the colors simply run together when rendered, but they are mathematically possible.)*

The bottom line on color

The color of each individual pixel is determined by the values stored in the three color bytes for that pixel. If you change any of those values, you will change the color of the pixel accordingly.

Now back to transparency

I'm going to explain transparency with an analogy. The alpha byte also has 256 possible values ranging from 0 to 255. If the value is zero, the pixel is completely transparent regardless of the values of the color bytes. If the value is 255, the pixel is completely opaque with the color of the pixel being determined exclusively by the values stored in the three color bytes.

What about the values between 0 and 255

Here is my analogy. Assume that you paint a glass window with purple paint that adheres well to glass. After the paint dries, what you will see when you look at the window is purple color. It is unlikely that you will see the green trees on the other side of the glass showing through the purple. This is what I mean by completely opaque.

Now assume that you paint the window with purple paint that doesn't adhere to glass very well. You will end up with a very thin coat of purple paint on the glass. When the paint dries, what you will see when you look at the window is a mixture of purple color and the green trees that show through from the other side. This represents an alpha value somewhere between 0 and 255. Whatever pixel was placed on the screen before the new pixel was placed there will show through the new pixel to some extent.

Now assume that you put cooking oil on the glass before you attempt to paint it and none of the purple paint sticks to the glass. Regardless of the fact that you attempted to apply purple paint to the window, what you see when you look at the window is the green trees on the other side of the glass. The purple doesn't show up at all. The window is completely transparent.

An alpha byte value of zero

This is what happens when the value of the alpha byte is 0. Whatever was there before is what you see even though the combination of red, green, and blue bytes is correct to cause the pixel to be purple. The pixel is purple. However, it is totally transparent so the purple color doesn't show.

> *(If you come back later and change the value of the alpha byte to a value between 0 and 255, the purple attribute of the pixel will become apparent.)*

Four bytes stored in an int value

What we are going to find is that once the image file has been read, and the pixel data has been extracted from the image, each pixel is represented by a four-byte array element of type **int**. However, at this point you shouldn't consider this to truly represent a value of type **int**. Rather, the array element of type **int** is simply a convenient place to pack four independent ***unsigned*** bytes end to end.

The order of the bytes

The most significant byte is the alpha byte. The next most significant is the red byte. The next byte is the green byte, and the least significant byte is the blue byte.

One of our tasks will be to extract the individual bytes from the **int** value and to get them out where we can easily manipulate them. We will use the bitwise operators **&** and >> to accomplish this.

Java doesn't support unsigned arithmetic. As a practical matter, it is very cumbersome to do arithmetic on unsigned byte data in Java. Therefore, we will extract each byte into an individual variable of type **int** to make it easier to do arithmetic involving the alpha and color values. In so doing, we will make certain that the bits that make up the unsigned color or alpha byte end up in the least significant eight bits of the variable, and the twenty-four other bits in the variable all have a value of 0.

Supplementary material

I recommend that you also study the other lessons in my extensive collection of online Java tutorials. You will find those lessons published at Gamelan.com. However, as of the date of this writing, Gamelan doesn't maintain a consolidated index of my Java tutorial lessons, and sometimes they are difficult to locate there. You will find a consolidated index at www.DickBaldwin.com.

Preview

Two programs and one interface

In this lesson, I will present and explain two Java programs and one Java interface. The two Java programs are named **ImgMod02** and **ProgramTest**. For convenience, these two programs are contained in the same source code file.

The program named **ImgMod02** is the framework or driver program. The program named **ProgramTest** is a simple image-processing program that is provided mainly as a test program for the driver program.

The interface is named **ImgIntfc02**. It is contained in a separate source code file.

The processImg method

The program named **ProgramTest**, *(and for that matter all image-processing programs that are capable of being driven by **ImgMod02**)*, must implement the interface named **ImgIntfc02**. The interface declares a single method named **processImg**, which must be defined by all implementing classes.

When the user runs the program named **ImgMod02**, that program instantiates an object of the image-processing program class and invokes the **processImg** method on that object. A three-dimensional array containing the pixel data for the image is passed to the method. The **processImg** method returns a three-dimensional array containing the pixel data for a modified version of the original image.

A *before* and *after* display

When the **processImg** method returns, the driver program causes the original image and the modified image to be displayed in a frame with the original image above the modified image *(see Figures 1 through 4 for examples of the display format).*

Default image-processing program and default image file

If the user doesn't specify an image-processing program, the driver program will instantiate an object of the class named **ProgramTest** and will invoke the **processImg** method on that object.

By default, the program will also search for an image file named **junk.gif** in the current directory, and will process that image file if it can be found

As mentioned earlier, the class definition for the **ProgramTest** class is included in the source code file along with the driver program. However, the image file named **junk.gif** must be provided by the user in the current directory.

> *(Just about any gif file of an appropriate size named **junk.gif** will do. You should make certain, however, that the image is small enough that two copies will fit on the screen when stacked one above the other.)*

The behavior of the ProgramTest program

The default image-processing program named **ProgramTest** draws a sloping white line across the image starting at the top left corner. A sample output produced by the image-processing program named **ProgramTest** with my image file named **junk.gif** is shown in Figure 5.

> *(You should be able to right-click on the image in Figure 6 and download and save the image locally in a file named **junk.gif**.)*

Figure 5 Default program output.

Controlling the slope of the line

The white line in Figure 5 has a default slope of 1.0. The program named **ProgramTest** provides an input dialog box that allows the user to specify the slope of the line.

To change the slope, the user types a new slope value into the text field in the dialog box *(not shown)* and clicks the **Replot** button at the bottom of Figure 5. This will cause the image to be reprocessed and the newly modified image will be displayed showing the line with the new slope.

> *(It isn't necessary to press the **Enter** key after typing the new slope value into the text field, but doing so won't cause any harm. Note that only positive slope values can be used. Entry of a negative slope value will cause an exception to be thrown.)*

Behavior with transparent areas

Other than to add the white line, the image-processing program named **ProgramTest** does not modify the image in any way. Note however that it does draw a visible white line across transparent areas, making the pixels that constitute the line non-transparent.

> *(The image in Figure 5 doesn't have any transparent areas.*
>
> *The frame produced by the driver program has a yellow background. You can see some of the background color showing between the two images in Figure 5. Note that it may be difficult to see the white line against the default yellow background when the line is drawn across a transparent area in an image.)*

Discussion and Sample Code

As mentioned earlier, this lesson presents and explains two programs named **ImgMod02** and **ProgramTest**. I will begin with a discussion of **ImgMod02**.

The program named ImgMod02

The purpose of this program is to make it easy to experiment with the modification of pixel data in an image and to display the modified version of the image along with the original version of the image.

Extracting and saving the pixel data

The program extracts the pixel data from an image file into a three-dimensional array of type:

```
int[row][column][color]
```

The first two dimensions of the array correspond to the rows and columns of pixels in the image. The number of rows and columns will be different from one image to the next.

The third dimension always has four elements. The elements along that axis contains the following values by index value:

- 0 alpha
- 1 red
- 2 green
- 3 blue

Data type

Note that the values in the three-dimensional array are stored as type **int** rather than type *unsigned byte*, which is the format of pixel data in the original image.

> *(Recall that type **byte** in Java is inherently signed. There is no such thing as an **unsigned byte** in Java. Furthermore, all arithmetic operations in Java are signed*

operations. As I recall, there is only one unsigned operation in Java, and that is an unsigned right shift operation.)

This conversion to type **int** eliminates many problems involving the requirement to perform arithmetic on unsigned byte data.

Image file types supported

The program supports **gif** and **jpg** files and possibly some other file types as well. If you are wondering about compatibility with a particular file type, just try using it. The program will abort with an error if the type isn't supported.

A framework or driver program

This program provides a framework that is designed to invoke another program to process the pixels extracted from an image.

In other words, this program extracts the pixels from an image and puts them in a format that is relatively easy to work with. A second program is then invoked to actually process the pixels.

The modified pixels are then returned to this program, which displays the modified image and the original image in the format shown in Figures 1 through 5.

Usage information

To use the program in its most versatile form, enter the following at the command line:

```
java ImgMod02 ProcessingProgramName ImageFileName
```

For test purposes, the source code includes a class definition for an image-processing program named **ProgramTest**.

Default image file is junk.gif

If the *ImageFileName* is omitted, the program will search for an image file in the current directory named **junk.gif** and will process it using the processing program specified by the second command-line argument.

Default processing program is ProgramTest

If both command-line arguments are omitted, the program will search for an image file in the current directory named **junk.gif** and will process it using the built-in processing program named **ProgramTest**.

Image file must be provided by the user

The image file must be provided by the user in all cases. However, it doesn't have to be in the current directory if a path to the file is specified on the command line. If the program is unable to load the image file within ten seconds, it will abort with an error message.

*(As mentioned earlier, you should be able to right-click on the image in Figure 6 and download and save the image locally under the file name **junk.gif**. Then you should be able to replicate the output produced in Figure 5 by omitting both the image-processing program name and the image file name.)*

Image display format

When the program is started, the original image and the processed image are displayed in a frame with the original image above the processed image.

A **Replot** button appears at the bottom of the frame. If the user clicks the **Replot** button, the image-processing method is rerun, the image is reprocessed and the new version of the processed image replaces the old version in the display.

Input to the image-processing program

The image-processing program may provide a GUI for data input making it possible for the user to modify the behavior of the image-processing method each time it is run. This capability is illustrated in the built-in processing program named **ProgramTest**.

The processImg method

The image-processing program must implement the interface named **ImgIntfc02**. A listing of that interface is provided in Listing 32. That interface declares a single method with the following signature:

```
int[][][] processImg(int[][][] threeDPix,
                     int imgRows,
                     int imgCols);
```

The first parameter is a reference to a three-dimensional array of pixel data stored as type **int**. The second and third parameters specify the number of rows of pixels and the number of columns of pixels in the image.

Parameterized constructor not supported

The image-processing program cannot have a parameterized constructor.

(More correctly, if the image-processing program has one or more parameterized constructors, they will simply be ignored.)

This is because an object of the image-processing class is instantiated by invoking the **newInstance** method of the class named **Class** on the name of the image-processing class provided as a **String** on the command line. This approach to object instantiation does not support parameterized constructors.

Similarly, if the image-processing program has a **main** method, it will also be ignored. Execution of code in the image-processing program is started by the invocation of the method named **processImg**.

More on the processImg method

The image-processing class must define the method named **processImg** with the signature given earlier. The **processImg** method receives a three-dimensional array containing pixel data. It should make a copy of the incoming array and modify the copy rather than modifying the original. Then the method should return a reference to the modified copy of the three-dimensional pixel array.

The method also receives the number of columns and the number of rows of pixels in the image represented by the three-dimensional array object.

Be careful of the range of values

The **processImg** method is free to modify the values of the pixels in the array in any manner before returning the modified array. Note however that native pixel data consists of four *unsigned* bytes.

If the modification of the pixel data produces negative values or positive value greater than 255, this should be dealt with before returning the modified pixel data. Otherwise, the returned values will simply be truncated to eight bits before display, and the result of displaying those truncated bits may not be as expected.

Dealing with out-of-range values

There are at least two ways to deal with this situation. One way is to simply clip all negative values at zero and to clip all values greater than 255 at 255.

The other way is to perform a further modification and map values in the range from -x to +y into the range from 0 to 255. With this approach, all the pixel values would be modified in the same way such that the minimum value contained in all the pixel color values is 0 and the maximum value is 255.

There is no one approach that is *right* approach for all situations.

Display both images

As described earlier, when the **processImg** method returns, the program named **ImgMod02** causes the original image and the modified image to be displayed in a frame on the screen with the original image being displayed above the modified image. Examples of this display format are shown in Figures 1 through 5.

Some operational details

This program reads an image file from the disk and saves it in memory under the name **rawImg**. Then it declares a one-dimensional array of type **int** of sufficient size to contain one **int** value for every pixel in the image.

(Each int value will later be populated with one alpha byte and three color bytes.)

The name of the one-dimensional array is **oneDPix**.

Then the program instantiates an object of type **PixelGrabber**, which associates the **rawImg** with the one-dimensional array of type **int**.

Following this, the program invokes the **grabPixels** method on the object of type **PixelGrabber** to cause the pixels in the **rawImg** to be extracted into **int** values and stored in the array named **oneDPix**.

Very similar to programs in earlier lessons

Down to this point, the program is very similar to programs that I explained in earlier lessons entitled Graphics - Introduction to Images and Graphics - Overview of Advanced Image Processing Capabilities. I will simply refer you to those lessons and won't repeat those explanations in this lesson.

Copy pixel values to three-dimensional int array

Then the program copies the pixel values from the **oneDPix** array into the **threeDPix** array, converting them to type **int** in the process. This is done for two reasons:

- To put the pixel data in a format that makes it easier to access for processing.
- To change the type from *unsigned byte* to **int** to eliminate the requirement to perform arithmetic on unsigned byte data.

The type conversion process involves some very special data handling to cause the unsigned pixel values to become positive values of type **int**. I will explain that later.

Process the image

The **threeDPix** array is passed to an image-processing program that is either specified on the command line or specified by a default class name. The image-processing program returns a modified version of the three-dimensional array of pixel data.

Create new image from modified pixel data

The **ImgMod02** program then creates a new version of the **oneDPix** array containing the modified pixel data. The program uses the **createImage** method of the **Component** class along with the constructor for the **MemoryImageSource** class to create a new image from the modified pixel data. The name of the new image is **modImg**.

Display the two images

Finally, the program overrides the **paint** method where it uses the **drawImage** method to display both the raw image and the modified image on the same **Frame** object. The raw image is displayed above the modified image with a very small amount of the background color of the frame showing between the two.

An ActionListener object

Along the way, the program registers an **ActionListener** object on the **Replot** button on the bottom of the **Frame**.

When the user clicks the **Replot** button, the listener object invokes the **processImg** method again on the image-processing object, passing the **threeDPix** array as a parameter. This causes the **processImg** method to once again process the incoming pixel data and to return a three-dimensional array containing modified pixel data.

The listener object then causes the modified image to be displayed below the original image.

If the user has provided new and different input information to the image-processing object before clicking the **Replot** button, the newly displayed modified image may be different from the previously displayed version of the modified image.

Testing

This program was tested using SDK 1.4.2 under WinXP.

Will discuss the program in fragments

As is my custom, I will break the program down and discuss it in fragments. A complete listing of the program is provided in Listing 31 near the end of the lesson.

The ImgMod02 class

The **ImgMod02** class definition begins in Listing 1.

```
class ImgMod02 extends Frame{
  Image rawImg;
```

```
  int imgCols;//Number of horizontal
pixels
  int imgRows;//Number of rows of
pixels
  Image modImg;//Reference to modified
image

  //Inset values for the Frame
  int inTop;
  int inLeft;

Listing 1
```

Note that the class extends **Frame**. In addition to being the control program, it is also the display GUI.

> *(In case you prefer the use of Swing components, you could easily use a **JFrame** instead by adding the invocation of **getContentPane** at the appropriate places.)*

Listing 1 declares several instance variables with descriptive names and comments. I will explain the use of these variables later.

Default image-processing program and image file

Listing 2 declares and initializes **String** instance variables with the names of the default image-processing program and the default image file name.

```
  static String theProcessingClass =

"ProgramTest";

  static String theImgFile =
"junk.gif";

Listing 2
```

As explained earlier, the default image file will be used if the user omits the name of the image file from the command line. The user must provide this image file in the current directory.

The default image-processing program will be used if the user omits both the name of the image file and the name of the image-processing program from the command line.

Additional instance variables

Listing 3 declares additional instance variables and initializes some of them.

```
  MediaTracker tracker;
```

```
  Display display = new Display();//A
Canvas
  Button replotButton = new
Button("Replot");

  //References to arrays that store
pixel data.
  int[][][] threeDPix;
  int[][][] threeDPixMod;
  int[] oneDPix;

  //Reference to the image-processing
object.
  ImgIntfc02 imageProcessingObject;
```

Listing 3

I will explain the use of the instance variables in Listing 3 later.

The main method

Listing 4 shows the beginning of the **main** method.

```
  public static void main(String[]
args){

    if(args.length == 0){
      //Use default processing class
and default
      // image file.  No code required
here.
      // Class and file names were
specified
      // above.  This case is provided
for
      // information purposes only.
    }else if(args.length == 1){
      theProcessingClass = args[0];
      //Use default image file
    }else if(args.length == 2){
      theProcessingClass = args[0];
      theImgFile = args[1];
    }else{
      System.out.println("Invalid
args");
      System.exit(1);
    }//end else
```

Listing 4

Listing 4 provides the logic to handle the command line arguments and implement the previously described behavior involving the default image file name and the default image-processing program.

This code in Listing 4 is straightforward and shouldn't require further explanation beyond the embedded comments.

Display name of image-processing program and image file

Listing 5 displays the name of the image-processing program and the image file to be processed.

```
    System.out.println("Processing
program: "
                           +
theProcessingClass);
    System.out.println("Image file: "
                                 +
theImgFile);

Listing 5
```

The code in Listing 5 is straightforward.

Instantiate an object of the ImgMod02 class

Listing 6 invokes the constructor for the ImgMod02 class to create an instance of the class.

```
    ImgMod02 obj = new ImgMod02();
  }//end main

Listing 6
```

Listing 6 also signals the end of the **main** method.

The constructor

The constructor begins in Listing 7. The code in Listing 7 gets an image from the specified image file. The file can be in a different directory from the current directory if the path to the file is provided on the command line along with the file name.

```
  public ImgMod02(){//constructor

    rawImg =
Toolkit.getDefaultToolkit().

getImage(theImgFile);
```

```
    //Use a MediaTracker object to
block until
    // the image is loaded or ten
seconds has
    // elapsed.
    tracker = new MediaTracker(this);
    tracker.addImage(rawImg,1);

    try{
      if(!tracker.waitForID(1,10000)){
        System.out.println("Load
error.");
        System.exit(1);
      }//end if
    }catch(InterruptedException e){
      e.printStackTrace();
      System.exit(1);
    }//end catch

    //Make certain that the file was
successfully
    // loaded.
    if((tracker.statusAll(false)
                      &
MediaTracker.ERRORED
                      &
MediaTracker.ABORTED) != 0){
      System.out.println(
                        "Load errored or
aborted");
      System.exit(1);
    }//end if
```

Listing 7

Code very similar to the code in Listing 7 was explained in previous lessons entitled Graphics - Introduction to Images and Graphics - Overview of Advanced Image Processing Capabilities. I will simply refer you to those lessons and won't repeat that explanation in this lesson.

Image has been loaded

At this point, the image contained in the image file has been loaded into memory. The code in Listing 8 begins by getting the width and the height of the image measured in pixels.

```
    imgCols = rawImg.getWidth(this);
    imgRows = rawImg.getHeight(this);

    this.setTitle("Copyright 2004,
Baldwin");
    this.setBackground(Color.YELLOW);
    this.add(display);
```

```
this.add(replotButton,BorderLayout.SOUTH);

Listing 8
```

In addition, the code in Listing 8:

- Sets some GUI properties including a yellow background color.
- Adds the display *(a **Canvas** object)* to the frame.
- Adds the **Replot** button to the bottom of the frame.

Set the Frame size

At this point, I need to set the size of the frame to accommodate the sizes of the images that will be displayed later. The code to accomplish this is shown in Listing 9.

```
    //Make it possible to get insets
and the
    // height of the button.
    setVisible(true);
    //Get and store inset data for the
Frame and
    // the height of the button.
    inTop = this.getInsets().top;
    inLeft = this.getInsets().left;
    int buttonHeight =

replotButton.getSize().height;
    //Size the frame
    this.setSize(2*inLeft+imgCols +
1,inTop
                 + buttonHeight +
2*imgRows + 7);

Listing 9
```

To begin with, I need to invoke the **getInsets** method on the frame in order to get the size of the banner at the top and the size of the borders on the sides and the bottom. In addition, I need to get the height of the **Replot** button at the bottom of the frame. I need to make the **Frame** large enough that a **Canvas**, which is twice the size of the image, will fit inside the banner, the borders, and the button.

Set visible property to true

In order to use **getInsets** to get the size of the banner and the borders, the **visible** property must be true. Therefore, the code in Listing 9 begins by invoking the **setVisible** method with a **true** parameter.

Then the code in Listing 9 invokes the appropriate methods to get and save the needed information.

Set the Frame size

Finally, the code in Listing 9 sets the size of the frame so that a small amount of the yellow background will show on the right and on the bottom when both images are displayed in the frame. Also, as you will see later, the placement of the images on the **Canvas** allows a small amount of the yellow background to show through between the two images.

(See Figure 5 for an example of the result of this sizing process.)

An ActionListener registered on the Replot button

Listing 10 shows the beginning of an anonymous inner class that is used to register an **ActionListener** object on the **Replot** button.

The **actionPerformed** method defined in the class is invoked whenever the user clicks the **Replot** button.

*(The method is also invoked once at startup when the program posts a counterfeit **ActionEvent** to the system event queue and attributes the event to the **Replot** button.)*

```
    replotButton.addActionListener(
      new ActionListener(){
        public void actionPerformed(

ActionEvent e){
          threeDPixMod =

imageProcessingObject.processImg(

threeDPix,imgRows,imgCols);
```

Listing 10

Behavior of the actionPerformed method

The **actionPerformed** method that begins in Listing 10 invokes the **processImg** method on the image-processing object, passing a reference to a three-dimensional array object containing pixel data. In addition, the number of rows and the number of columns of pixels in the image represented by the three-dimensional array object are passed as parameters.

The **processImg** method processes the image and returns a three-dimensional array object containing pixel data for a modified version of the image.

Convert pixel data back to a one-dimensional array

Later we will see that the **grabPixels** method of the **PixelGrabber** class is used to extract the pixel data from the image into a one-dimensional array of type **int**. Then we will see that a method of my own design is used to copy that pixel data into a three-dimensional array that is more suitable for processing. That is the three-dimensional array that is passed to the **processImg** method in Listing 10.

In order to display the modified image, we need to populate a new object of type **Image** with the modified pixel data. In order to do that, we need to convert the pixel data back into a one-dimensional array of type **int** in the same format originally produced by the **grabPixels** method.

This seems like as good a time as any to explain the formats of the one-dimensional and three-dimensional arrays that contain pixel data.

Format of the one-dimensional pixel array

The **grabPixels** method of the **PixelGrabber** class extracts the pixel data from an **Image** object into a one-dimensional array of type **int**.

Each element in the array contains the four unsigned data bytes that represent a single pixel. The most significant byte contains the alpha or transparency data. Moving from most to least significant, the remaining bytes contain the unsigned values for red, green, and blue in that order.

If the image has N columns of pixels in each row, the first N elements in the array contain the data for the first row; the second N elements contain the data for the second row, etc.

Format of the three-dimensional pixel array

To make it easier to process the pixel data, a method named **convertToThreeDim** is used to extract the pixel data from the one-dimensional array and to populate a three-dimensional array of type **int**. Each **byte** in the one-dimensional array is used to populate the bottom eight bits of an element of type **int** in the three-dimensional array.

The dimensions of the three-dimensional array are *row*, *col*, and *color* in that order. Row and col correspond to the rows and columns of the image measured in pixels. Color corresponds to transparency and color information at the following index levels in the third dimension:

- 0 alpha
- 1 red
- 2 green
- 3 blue

I will set the discussion of the **actionPerformed** method aside for the time being and explain the two methods that are used to convert the pixel data back and forth between the one-dimensional and three-dimensional array formats.

The convertToThreeDim method

The **convertToThreeDim** method begins in Listing 11.

```
  int[][][] convertToThreeDim(
          int[] oneDPix,int
imgCols,int imgRows){
    //Create the new 3D array to be
populated
    // with color data.
    int[][][] data =
                     new
int[imgRows][imgCols][4];
```

Listing 11

The purpose of this method is to convert the data in the one-dimensional **int** array *(containing pixel data of type unsigned byte)* into a three-dimensional array of type **int**.

The structure of the code in this method is determined by the way the pixel data is formatted into the one-dimensional array of type **int** produced by the **grabPixels** method of the **PixelGrabber** object.

Create an empty three-dimensional array object

The code in Listing 11 begins by creating a new empty three-dimensional array object that will be populated with pixel data. The data corresponding to a specific pixel in the three-dimensional array will be located at the intersection of a specific row and a specific column. That data will consist of four elements of type **int** where the least significant eight bits of each **int** element are populated with the eight bits corresponding to an eight-bit unsigned color or alpha value. The most significant twenty-four bits of each element will be set to 0.

When populating the elements in this array, care will be taken to ensure that the unsigned eight-bit values are not allowed to sign-extend into the upper twenty-four bits of the **int** value, even for unsigned byte values of **0x80** and higher.

Extract each row of pixel data

The algorithm for extracting the pixel data from the one-dimensional array and populating the three-dimensional array handles one row of pixels at a time.

Listing 12 shows the beginning of a **for** loop that iterates once for each row of pixels in the image.

```
    for(int row = 0;row <
imgRows;row++){
```

```
      //Extract a row of pixel data
into a
      // temporary array of ints
      int[] aRow = new int[imgCols];
      for(int col = 0; col <
imgCols;col++){
        int element = row * imgCols +
col;
        aRow[col] = oneDPix[element];
      }//end for loop on col
```

Listing 12

Listing 12 begins by extracting one row of pixel data from the original one-dimensional array that contains the entire image into a new temporary one-dimensional array that contains the pixel data for one row only.

The code in Listing 12 contains an inner **for** loop that iterates on columns. Each time that loop terminates, the one-dimensional array referred to by **aRow** contains one element of type **int** for each pixel in the row. Each element of type **int** contains one alpha byte and there color bytes in the packed format described earlier.

Populate the three-dimensional array for each row

The code in Listing 13 contains a **for** loop that iterates once for each pixel in the row.

```
      //Move the data into the 3D
array.  Note
      // the use of bitwise AND and
bitwise right
      // shift operations to mask all
but the
      // correct set of eight bits.
      for(int col = 0;col <
imgCols;col++){
        //Alpha data
        data[row][col][0] = (aRow[col]
>> 24)

& 0xFF;
        //Red data
        data[row][col][1] = (aRow[col]
>> 16)

& 0xFF;
        //Green data
        data[row][col][2] = (aRow[col]
>> 8)

& 0xFF;
        //Blue data
```

```
        data[row][col][3] =
(aRow[col])

& 0xFF;
      }//end for loop on col
    }//end for loop on row
    return data;
  }//end convertToThreeDim

Listing 13
```

During each iteration, the code in Listing 13 shifts each unsigned byte of interest into the least significant eight bits of an **int**. Then it performs a bitwise ***and*** operation with the hexadecimal value **0xFF** to force the twenty-four most significant bits to have a value of zero and to preserve the values of the eight least-significant bits. This operation is performed once for the alpha byte and once for each of the three color bytes.

The end result

The end result is that each of the four color/alpha elements of type **int** at the intersection of a specific row and a specific column consists of an **int** element with zeros in the twenty-four most significant bits and the values of the bits from the original unsigned byte in the eight least-significant bits. The four elements at the intersection contain the values for the alpha byte and the three color bytes for a single pixel.

The convertToOneDim method

We started down this side trip in our discussion of the inner class used to register an **ActionListener** object on the **Replot** button. We had reached the point in that discussion where we were just about to say that the following code converts the modified pixel data in the three-dimensional array back into a one-dimensional array so that it can be used to create a new object of type **Image**.

At that point, we decided to explain the format of the one-dimensional array and the three-dimensional array and to discuss the methods used to convert the data back and forth between the two.

At this point, you understand the format of the one-dimensional array produced by the **grabPixels** method of the **PixelGrabber** class, and you understand the format of the three-dimensional array produced by the method named **convertToThreeDim**.

Listing 14 contains the entire method named **convertToOneDim**.

```
  int[] convertToOneDim(
          int[][][] data,int
imgCols,int imgRows){
    //Create the 1D array of type int
```

```
to be
    // populated with pixel data, one
int value
    // per pixel, with four color and
alpha bytes
    // per int value.
    int[] oneDPix = new int[
                            imgCols *
imgRows * 4];

    //Move the data into the 1D array.
Note the
    // use of the bitwise OR operator
and the
    // bitwise left-shift operators to
put the
    // four 8-bit bytes into each int.
    for(int row = 0,cnt = 0;row <
imgRows;row++){
      for(int col = 0;col <
imgCols;col++){
        oneDPix[cnt] =
((data[row][col][0] << 24)
                                       &
0xFF000000)
                     |
((data[row][col][1] << 16)
                                       &
0x00FF0000)
                      |
((data[row][col][2] << 8)
                                       &
0x0000FF00)
                               |
((data[row][col][3])
                                       &
0x000000FF);
        cnt++;
      }//end for loop on col

    }//end for loop on row

    return oneDPix;
  }//end convertToOneDim
}//end ImgMod02.java class
```

Listing 14

Reverse the process

The purpose of the **convertToOneDim** method is to convert the data in the three-dimensional array of type back into a one-dimensional array of type int in the same format as that produced by the **grabPixels** method of the **PixelGrabber** class.

This is the reverse of the process implemented by the method named **convertToThreeDim**. If you understand the use of bitwise operators, the code in Listing 14 shouldn't require further explanation. If not, you can learn a little about bitwise operators in the earlier lesson entitled Operators.

Back to the actionPerformed method

Now it's time to pick back up with the discussion of the **actionPerformed** method where we left off with Listing 10. I had just explained that in order to display the modified image based on the pixel data received in Listing 10, we need to convert that data back into a one-dimensional array in the format produced by the **grabPixels** method.

Listing 15 picks up with the next statement in the **actionPerformed** method.

Convert pixel data to a one-dimensional array format

The statement in Listing 15 invokes the **convertToOneDim** method to convert the modified pixel data in the three-dimensional array back into a one-dimensional array in the correct format.

```
//In the actionPerformed method

          oneDPix = convertToOneDim(

threeDPixMod,imgCols,imgRows);

Listing 15
```

Create a new Image object

The statement in Listing 16 uses the **createImage** method and the **MemoryImageSource** class to create a new object of type **Image**, referred to by the variable named **modImg**.

> *(Once again, I explained this procedure in the earlier lesson entitled Graphics - Overview of Advanced Image Processing Capabilities. You can read an explanation there.)*

```
          //Use the createImage()
method to
          // create a new image from
the 1D array
          // of pixel data.
          modImg = createImage(
             new MemoryImageSource(

imgCols,imgRows,oneDPix,0,imgCols));

Listing 16
```

Repaint the display

Finally, the code in Listing 17 requests a repaint on the **Canvas** object. This causes the overridden **paint** method of the **Display** class to be invoked to draw the original image and the modified image in the frame on the screen.

```
          display.repaint();
        }//end actionPerformed
      }//end ActionListener
    );//end addActionListener
    //End anonymous inner class.

Listing 17
```

We will see the code for the overridden **paint** method a little later.

Listing 7 signals the end of the definition of the anonymous inner class.

Now back to the constructor

Now it's time to pick back up with the discussion of the constructor where we left off in Listing 9, before getting involved in the discussion of the inner class for the **ActionListener** object.

Create a one-dimensional array object

The code in Listing 18 creates an empty one-dimensional array object that will later be populated with pixel data produced by the **grabPixels** method. These pixels will represent the original image.

```
    oneDPix = new int[imgCols *
imgRows];

Listing 18
```

Get the pixel data from the original image

The code in Listing 19 extracts the pixel data from the original image and stores it in the one-dimensional array created in Listing 18.

```
    //Convert the rawImg to numeric
pixel
    // representation.  Note that
grapPixels()
    // throws InterruptedException
    try{
```

```
      //Instantiate a PixelGrabber
object
      // specifying oneDPix as the
array in which
      // to put the numeric pixel
data. See Sun
      // docs for parameters
      PixelGrabber pgObj = new
PixelGrabber(

rawImg,0,0,imgCols,imgRows,

oneDPix,0,imgCols);
      //Invoke the grabPixels() method
on the
      // PixelGrabber object to
extract the pixel
      // data from the image into an
array of
      // numeric pixel data stored in
oneDPix.
      // Also test for success in the
process.
      if(pgObj.grabPixels() &&

((pgObj.getStatus() &

ImageObserver.ALLBITS)

!= 0)){
```

Listing 19

You can read an explanation of this code in the earlier lesson entitled Graphics - Overview of Advanced Image Processing Capabilities.

Convert pixel data to three-dimensional format

At this point, the pixel data has been extracted from the original image and has been stored in a one-dimensional array. It is time to convert that pixel data to the three-dimensional array format that provides for ease of access while processing the pixels.

This is accomplished by the code in Listing 20, which invokes the **convertToThreeDim** method to make the conversion.

```
        threeDPix = convertToThreeDim(

oneDPix,imgCols,imgRows);
```

Listing 20

Instantiate an image-processing object

At this point, we have captured the pixel data from the original image in a three-dimensional array suitable for processing. All we lack is an object to do the processing.

The statement in Listing 21 invokes the **newInstance** method of the class named **Class** to instantiate an object of the image-processing class whose name was provided as a **String** in a command-line parameter.

> *(The name of the image-processing class may also have been obtained by default if the user failed to specify the class name on the command line.)*

```
        try{
          imageProcessingObject = (

ImgIntfc02)Class.forName(

theProcessingClass).newInstance();
```

Listing 21

The use of the class named Class

If you are unfamiliar with this approach to the instantiation of objects, you can learn about it in the earlier lesson entitled More on Inheritance. You will also find examples of the use of this approach in the lesson entitled The Essence of OOP using Java, Array Objects, Part 3, as well as in numerous other lessons on my website.

Note that this approach does not support the use of parameterized constructors.

Fire an ActionEvent

At this point, we have the pixel data in the correct format and we have an image-processing object that will process those pixels and return an array containing modified pixel values. All we need to do is to invoke the **processImg** method on the image-processing object passing the pixel data and other appropriate information as parameters.

This is accomplished in Listing 22.

```
          Toolkit.getDefaultToolkit().

getSystemEventQueue().postEvent(
                  new ActionEvent(
                    replotButton,

ActionEvent.ACTION_PERFORMED,
```

```
                    "Replot"));

Listing 22
```

Post a counterfeit ActionEvent to the system event queue

Listing 22 posts a counterfeit **ActionEvent** to the system event queue and attributes the event to the **Replot** button. The result is exactly the same as if the user had clicked the **Replot** button. In either case, the **actionPerformed** method *(see Listing 10)* is executed.

> *(If you are unfamiliar with the use of the system event queue, you can learn about it in the earlier lesson entitled Posting Synthetic Events to the System Event Queue.)*

Invoke the processImg method

Referring back to the **actionPerformed** method in Listing 10, we see that posting this event causes the image-processing method named **processImg to** be invoked, passing the three-dimensional array of pixel data to the method, and receiving a three-dimensional array of modified pixel data back from the method.

Create and display a modified image

Referring back to Listings 15, 16, and 17, we also see that the **actionPerformed** method causes the modified pixel data to be used to create a new **Image** object, and causes that new image object to be displayed, along with the original image in the frame.

The first image-processing pass is complete

At this point, the image has been processed and both the original image and the modified image have been displayed. From this point forward, each time the user clicks the **Replot** button, the image will be processed again and the new modified image will be displayed along with the original image.

Remaining constructor code

The remaining code in the constructor is completely straightforward, so I won't discuss it further. You can view that code in Listing 31 near the end of the lesson.

The Display class

An object of the inner class shown in Listing 23 is a **Canvas** object upon which the two images are drawn.

> *(Recall that an object of the **Display** class was added to the client area of the **Frame** object in Listing 8. The **Display** class extends the **Canvas** class.)*

```
  class Display extends Canvas{
    public void paint(Graphics g){
      //First confirm that the image
has been
      // completely loaded and neither
image
      // reference is null.
      if (tracker.statusID(1, false)
==

MediaTracker.COMPLETE){
        if((rawImg != null)  &&
                                  (modImg
!= null)){

g.drawImage(rawImg,0,0,this);
          g.drawImage(modImg,0,imgRows
+ 1,this);
        }//end if
      }//end if
    }//end paint()
  }//end class myCanvas

Listing 23
```

Overriding the paint method

Listing 23 overrides the **paint** method, causing the two images to be drawn on the canvas, which is the standard way of drawing on a **Canvas** component in Java. *(I have used and discussed overridden **paint** methods in numerous earlier lessons.)*

The overridden **paint** method is invoked whenever there is a requirement to repaint the canvas. This can occur for a variety of reasons, including the invocation of the **repaint** method in the **ActionEvent** handler in Listing 17.

With the possible exception of the code involving the **MediaTracker**, the code in Listing 23 is straightforward and shouldn't require further discussion.

> *(Once again, you can learn about **MediaTracker** in the earlier lessons entitled Graphics - Introduction to Images and Graphics - Overview of Advanced Image Processing Capabilities. I will simply refer you to those lessons and won't repeat that explanation in this lesson.)*

End of ImgMod02 program

That completes the discussion of the **ImgMod02** program, and brings us to the topic of the built-in image-processing program named **ProgramTest**.

Fortunately, the **ProgramTest** program is much simpler than the **ImgMod02** program. This is fortunate because in order to experiment with different image-processing algorithms, you only need to replace the program named **ProgramTest**. You can use the program named **ImgMod02**, as written, with no changes.

The program named ProgramTest

The purpose of this program class is to provide a simple example of an image-processing class that is compatible with the program named **ImgMod02**.

The constructor for the class displays a small **Frame** on the screen with a single **TextField** object. The purpose of the text field is to allow the user to enter a value that represents the slope of a line.

In operation, the user types a value into the text field and then clicks the **Replot** button on the main image display frame. The user is not required to press the **Enter** key after typing the new value, but it doesn't do any harm to do so.

The processImg method

The class defines a method named **processImg** with the signature shown in Listing 32 near the end of the lesson.

The method named **processImg** receives a three-dimensional array containing alpha, red, green, and blue pixel values for an image. The values are received as type **int** *(not type byte).*

Copy and modify the pixels

A copy of the **threeDPix** array is made and saved. The copy is then modified to cause a white diagonal line to be drawn down and to the right from the upper left corner of the image when the modified pixels are used to create and display a new image. The three-dimensional array containing the modified pixel data is then returned to the calling method named **actionPerformed** in the program named **ImgMod02**.

The initial slope is 1.0

The first time the **processImg** method is invoked, the slope of the line has a value of 1.0. Thereafter, the slope of the line is controlled by a value that is typed into the text field prior to clicking the **Replot** button. Note that negative slope values will cause the program to throw an exception and abort.

The image is not modified in any way other than to draw the sloping white line on the image.

To cause a new line to be drawn, type a slope value into the text field and click the **Replot** button at the bottom of the image display frame.

Extends Frame and implements ImgIntfc02

This class extends **Frame** because it provides a GUI for user data input. However, a class that is compatible with **ImgMod02** is not required to extend the **Frame** class.

A compatible class is required, however, to implement the interface named **ImgIntfc02**. A listing of the interface is shown in Listing 32.

The class definition for ProgramTest

The beginning of the **ProgramTest** class, including the declaration of some instance variables is shown in Listing 24.

```
class ProgramTest extends Frame
                           implements
ImgIntfc02{

  double slope;//Controls the slope of
the line
  String inputData;//Obtained via the
TextField
  TextField inputField;//Reference to
TextField
```

Listing 24

The constructor

The entire constructor is shown in Listing 25.

> *(Recall that the constructor must not require any parameters in order to be compatible with the program named **ImgMod02**.)*

```
  //Constructor must take no
parameters
  ProgramTest(){
    //Create and display the user-
input GUI.
    setLayout(new FlowLayout());

    Label instructions = new Label(
              "Type a slope value and
Replot.");
    add(instructions);

    inputField = new
TextField("1.0",5);
    add(inputField);
```

```
    setTitle("Copyright 2004,
Baldwin");
    setBounds(400,0,200,100);
    setVisible(true);
  }//end constructor

Listing 25
```

Initial value in the text field

The code in the constructor is completely straightforward and should not require any further discussion. However, I will point out that the new **TextField** object is instantiated with an initial value of "1.0" for the **text** property. This is what causes the line to have a slope of 1.0 the first time the **processImg** method is invoked *(before the user has an opportunity to change the value).*

The processImg method

Because this class implements the **ImgIntfc02** interface, and because that interface declares the **processImg** method, this class must define the method.

The beginning of the **processImg** method is shown in Listing 26.

```
  public int[][][] processImg(
                              int[][][]
threeDPix,
                              int
imgRows,
                              int
imgCols){

    System.out.println("Program
test");
    System.out.println("Width = " +
imgCols);
    System.out.println("Height = " +
imgRows);

Listing 26
```

Note that in addition to receiving a three-dimensional array containing the pixel data for the image, the method also receives the number of rows and the number of columns of pixels in the image.

The code in Listing 26 simply displays some interesting information about the image being processed.

Make a working copy

The code in Listing 27 makes a working copy of the three-dimensional array of pixel data in order to avoid making permanent changes to the original image data.

```
    int[][][] temp3D =
                      new
int[imgRows][imgCols][4];

    for(int row = 0;row <
imgRows;row++){
      for(int col = 0;col <
imgCols;col++){
        temp3D[row][col][0] =

threeDPix[row][col][0];
        temp3D[row][col][1] =

threeDPix[row][col][1];
        temp3D[row][col][2] =

threeDPix[row][col][2];
        temp3D[row][col][3] =

threeDPix[row][col][3];
      }//end inner loop
    }//end outer loop

Listing 27
```

Get the slope value

Listing 28 gets the slope value from the text field as type **String** and converts it to type **double**.

```
    slope = Double.parseDouble(

inputField.getText());

Listing 28
```

The equation for a straight line

You may recall from your analytical geometry class that the equation for a straight line in Cartesian coordinates is:

y = m * x + b

where:
m is the slope of the line
b is the intersection of the line with the y-axis

In our case, the intersection value is 0 and the slope is the value obtained from the text field.

You may also remember that when drawing graphics on a computer screen using Java, the positive direction for the y-axis is down the screen. Thus, a line with a positive slope will go down and to the right.

Draw a white line on the working copy of the image data

Listing 29 modifies the pixel values such that when the modified pixels are rendered as an object of type **Image**, a white line will originate at the upper left corner of the image and proceed down and to the right.

```
    for(int col = 0;col <
imgCols;col++){
      int row = (int)(slope*col);
      if(row > imgRows -1)break;
      //Set values for alpha, red,
green, and
      // blue colors.
      temp3D[row][col][0] =
(byte)0xff;
      temp3D[row][col][1] =
(byte)0xff;
      temp3D[row][col][2] =
(byte)0xff;
      temp3D[row][col][3] =
(byte)0xff;
    }//end for loop
```

Listing 29

Equivalent variables

In Listing 29, the variable **col** is equivalent to the variable **x** in the equation of a straight line given earlier. Similarly, the variable **row** is equivalent to the variable **y** in that equation. The variable **slope** is equivalent to the variable **m** in that equation.

The value of **slope** used to relate each pixel on the x-axis *(col)* to a specific pixel on the y-axis *(row)*.

Once that relationship has been determined for each value of **col**, the alpha value and each of the three color values for the pixel at the intersection of that value of **col** and that value of **row** is set to **0xff** *(255)*. The new values replace the previous alpha and color values for that pixel.

Setting the alpha value to **0xff** causes the pixel to be completely opaque. Setting all three color values to **0xff** causes the combination of those three color contributions to result in a pixel that appears white to a human observer, as shown in Figure 5.

Return the modified pixel data

The statement in Listing 30 returns a reference to the modified three-dimensional array of pixel data to the calling method in the program named **ImgMod02**, where it will used to create and display a new image at the bottom of the frame.

```
    return temp3D;
  }//end processImg
}//end class ProgramTest

Listing 30
```

Listing 30 also signals the end of the **processImg** method and the end of the **ProgramTest** class.

That's all there is to it

As you can see, once you have the program named **ImgMod02** to handle all of the hard work, all that's required to create and test a new image-processing algorithm is to define a new class that:

- Makes a copy of an incoming three-dimensional array of pixel data representing an image.
- Modifies the pixel values in the copy according to some algorithm of your own design.
- Returns a reference to the modified three-dimensional array of pixel data for display.

Some cautions

A couple of cautions are probably in order. One caution is to beware of transparency. You should make certain that you don't end up with a modified array in which all the alpha values are zero.

> *(Recall that the elements in a new array of type **int** are automatically initialized to zero, so this is an easy mistake to make.)*

If you do, then your modified image will be completely transparent regardless of what you did to the color values for the pixels. As a result, the display will simply show the yellow background color for the frame.

Value ranges

Another caution has to do with the range of alpha and color values associated with the pixels. None of the values should be negative, and none of the values should exceed +255. If your values don't comply with these limits, the display will probably not be what you expect to see.

Prior to display, each of the four pixel values of type **int** will be converted to eight bits by simply discarding all but the least significant eight bits in each **int** element in the three-dimensional

array. This is not the same as clipping the values at 0 and 255, and will probably lead to unexpected results.

Run the Program

I encourage you to copy, compile, and run the program provided in this lesson. Experiment with it, making changes and observing the results of your changes. Above all, have fun and learn as much as you can about modifying image pixels using Java.

You should be able to right-click on the image in Figure 6 and download and save it locally under the name **junk.gif**. Then you should be able to replicate the output produced in Figure 5.

Figure 6 Raw image for junk.gif

Summary

I showed you how to modify an image by modifying the pixels belonging to that image. I also provided a program that makes it easy to modify the pixels in an image and to display the modified image.

What's Next?

Future lessons will show you how to write image-processing programs that implement many common special effects as well as a few that aren't so common. This will include programs to do the following:

- Highlight a particular area in an image.
- Blur all or part of an image.
- Sharpen all or part of an image.
- Perform edge detection on an image.
- Apply color filtering to an image.
- Apply color inversion to an image.

- Morphing one image into another image.
- Rotating an image.
- Squeezing part of an image into a smaller size.
- Controlling the brightness of an image using linear and non-linear algorithms.
- Other special effects that I may dream up or discover while doing the background research for the lessons in this series.

Complete Program Listing

Complete listings of the program and interface discussed in this lesson are provided below.

```
/*File ImgMod02.java.java
Copyright 2004, R.G.Baldwin

The purpose of this program is to make it easy
to experiment with the modification of pixel
data in an image and to display the modified
version of the image along with the original
version of the image.

The program extracts the pixel data from an
image file into a 3D array of type:

int[row][column][depth].

The first two dimensions of the array correspond
to the rows and columns of pixels in the image.
The third dimension always has a value of 4 and
contains the following values by index value:

0 alpha
1 red
2 green
3 blue

Note that these values are stored as type int
rather than type unsigned byte which is the
format of pixel data in the original image.
This type conversion eliminates many problems
involving the requirement to perform unsigned
arithmetic on unsigned byte data.

The program supports gif and jpg files and
possibly some other file types as well.

Operation:  This program provides a framework
that is designed to invoke another program to
process the pixels extracted from an image.
In other words, this program extracts the pixels
and puts them in a format that is relatively
easy to work with.  A second program is invoked
to actually process the pixels.  Typical usage
```

is as follows:

```
java ImgMod02 ProcessingProgramName ImageFileName
```

For test purposes, the source code includes a class definition for an image-processing program named ProgramTest.

If the ImageFileName is not specified on the command line, the program will search for an image file in the current directory named junk.gif and will process it using the processing program specified by the second command-line argument.

If both command-line arguments are omitted, the program will search for an image file in the current directory named junk.gif and will process it using the built-in processing program named ProgramTest.

The image file must be provided by the user in all cases. However, it doesn't have to be in the current directory if a path to the file is specified on the command line.

When the program is started, the original image and the processed image are displayed in a frame with the original image above the processed image. A Replot button appears at the bottom of the frame. If the user clicks the Replot button, the image-processing method is rerun, the image is reprocessed and the new version of the processed image replaces the old version.

The processing program may provide a GUI for data input making it possible for the user to modify the behavior of the image-processing method each time it is run. This capability is illustrated in the built-in processing program named ProgramTest.

The image-processing programming must implement the interface named ImgIntfc02. That interface declares a single method with the following signature:

```
int[][][] processImg(int[][][] threeDPix,
                     int imgRows,
                     int imgCols);
```

The first parameter is a reference to the 3D array of pixel data stored as type int. The last two parameters specify the number of rows of pixels and the number of columns of pixels in the image.

The image-processing program cannot have a parameterized constructor. This is because an object of the class is instantiated by invoking the newInstance method of the class named Class on the name of the image-processing program provided as a String on the command line. This approach to object instantiation does not support parameterized constructors.

If the image-processing program has a main method, it will be ignored.

The processImg method receives a 3D array containing pixel data. It should make a copy of the incoming array and modify the copy rather than modifying the original. Then the program should return a reference to the modified copy of the 3D pixel array.

The program also receives the width and the height of the image represented by the pixels in the 3D array.

The processImg method is free to modify the values of the pixels in the array in any manner before returning the modified array. Note however that native pixel data consists of four unsigned bytes. If the modification of the pixel data produces negative values or positive value greater than 255, this should be dealt with before returning the modified pixel data. Otherwise, the returned values will simply be masked to eight bits before display, and the result of displaying those masked bits may not be as expected.

There are at least two ways to deal with this situation. One way is to simply clip all negative values at zero and to clip all values greater than 255 at 255. The other way is to perform a further modification so as to map the range from -x to +y into the range from 0 to 255. There is no one correct way for all situations.

When the processImg method returns, this program causes the original image and the modified image to be displayed in a frame on the screen with the original image above the modified image.

If the user doesn't specify an image-processing program, this program will instantiate and use an object of the class named ProgramTest and an image file named junk.gif. The class definition for the ProgramTest class is included in this source code file. The image file named junk.gif

must be provided by the user in the current directory. Just about any gif file of an appropriate size will do. Make certain that it is small enough so that two copies will fit on the screen when stacked one above the other.

The processing program named ProgramTest draws a diagonal white line across the image starting at the top left corner. The program provides a dialog box that allows the user to specify the slope of the line. To change the slope, type a new slope into the text field and press the Replot button on the main graphic frame. It isn't necessary to press the Enter key after typing the new slope value into the text field, but doing so won't cause any harm. (Note that only positive slope values can be used. Entry of a negative slope value will cause an exception to be thrown.)

Other than to add the white line, the image processing program named ProgramTest does not modify the image. It does draw a visible white line across transparent areas, making the pixels underneath the line non-transparent. However, it may be difficult to see the white line against the default yellow background in the frame.

If the program is unable to load the image file within ten seconds, it will abort with an error message.

Some operational details follow.

This program reads an image file from the disk and saves it in memory under the name rawImg. Then it declares a one-dimensional array of type int of sufficient size to contain one int value for every pixel in the image. Each int value will be populated with one alpha byte and three color bytes. The name of the array is oneDPix.

Then the program instantiates an object of type PixelGrabber, which associates the rawImg with the one-dimensional array of type int. Following this, the program invokes the grabPixels method on the object of type PixelGrabber to cause the pixels in the rawImg to be extracted into int values and stored in the array named oneDPix.

Then the program copies the pixel values from the oneDPix array into the threeDPix array, converting them to type int in the process. The threeDPix array is passed to an image-processing

```
program.

The image-processing program returns a modified
version of the 3D array of pixel data.

This program then creates a new version of the
oneDPix array containing the modified pixel data.
It uses the createImage method of the Component
class along with the constructor for the
MemoryImageSource class to create a new image
from the modified pixel data.  The name of the
new image is modImg.

Finally, the program overrides the paint method
where it uses the drawImage method to display
both the raw image and the modified image on the
same Frame object.  The raw image is displayed
above the modified image.

Tested using SDK 1.4.2 under WinXP.
*********************************************/
import java.awt.*;
import java.awt.event.*;
import java.awt.image.*;

class ImgMod02 extends Frame{
  Image rawImg;
  int imgCols;//Number of horizontal pixels
  int imgRows;//Number of rows of pixels
  Image modImg;//Reference to modified image

  //Inset values for the Frame
  int inTop;
  int inLeft;

  //Default image-processing program.  This
  // program will be executed to process the
  // image if the name of another program is not
  // entered on the command line.  Note that the
  // class file for this program is included in
  // this source code file.
  static String theProcessingClass =
                                  "ProgramTest";

  //Default image file name.  This image file
  // will be processed if another file name is
  // not entered on the command line.  You must
  // provide this file in the current directory.
  static String theImgFile = "junk.gif";

  MediaTracker tracker;
  Display display = new Display();//A Canvas
  Button replotButton = new Button("Replot");

  //References to arrays that store pixel data.
  int[][][] threeDPix;
```

```
  int[][][] threeDPixMod;
  int[] oneDPix;

  //Reference to the image-processing object.
  ImgIntfc02 imageProcessingObject;
  //----------------------------------------//

  public static void main(String[] args){

    //Get names for the image-processing program
    // and the image file to be processed.
    // Program supports gif files and jpg files
    // and possibly some other file types as
    // well.
    if(args.length == 0){
      //Use default processing class and default
      // image file.  No code required here.
      // Class and file names were specified
      // above.  This case is provided for
      // information purposes only.
    }else if(args.length == 1){
      theProcessingClass = args[0];
      //Use default image file
    }else if(args.length == 2){
      theProcessingClass = args[0];
      theImgFile = args[1];
    }else{
      System.out.println("Invalid args");
      System.exit(1);
    }//end else

    //Display name of processing program and
    // image file.
    System.out.println("Processing program: "
                           + theProcessingClass);
    System.out.println("Image file: "
                                 + theImgFile);

    //Instantiate an object of this class
    ImgMod02 obj = new ImgMod02();
  }//end main
  //----------------------------------------//

  public ImgMod02(){//constructor
    //Get an image from the specified file.  Can
    // be in a different directory if the path
    // was entered with the file name on the
    // command line.
    rawImg = Toolkit.getDefaultToolkit().
                            getImage(theImgFile);

    //Use a MediaTracker object to block until
    // the image is loaded or ten seconds has
    // elapsed.
    tracker = new MediaTracker(this);
    tracker.addImage(rawImg,1);
```

```
    try{
      if(!tracker.waitForID(1,10000)){
        System.out.println("Load error.");
        System.exit(1);
      }//end if
    }catch(InterruptedException e){
      e.printStackTrace();
      System.exit(1);
    }//end catch

    //Make certain that the file was successfully
    // loaded.
    if((tracker.statusAll(false)
                  & MediaTracker.ERRORED
                  & MediaTracker.ABORTED) != 0){
      System.out.println(
                      "Load errored or aborted");
      System.exit(1);
    }//end if

    //Raw image has been loaded.  Get width and
    // height of the raw image.
    imgCols = rawImg.getWidth(this);
    imgRows = rawImg.getHeight(this);

    this.setTitle("Copyright 2004, Baldwin");
    this.setBackground(Color.YELLOW);
    this.add(display);
    this.add(replotButton,BorderLayout.SOUTH);
    //Make it possible to get insets and the
    // height of the button.
    setVisible(true);
    //Get and store inset data for the Frame and
    // the height of the button.
    inTop = this.getInsets().top;
    inLeft = this.getInsets().left;
    int buttonHeight =
                   replotButton.getSize().height;
    //Size the frame so that a small amount of
    // yellow background will show on the right
    // and on the bottom when both images are
    // displayed, one above the other.  Also, the
    // placement of the images on the Canvas
    // causes a small amount of background to
    // show between the images.
    this.setSize(2*inLeft+imgCols + 1,inTop
                 + buttonHeight + 2*imgRows + 7);

    //=======================================//
    //Anonymous inner class listener for replot
    // button.  This actionPerformed method is
    // invoked when the user clicks the Replot
    // button.  It is also invoked at startup
    // when this program posts an ActionEvent to
    // the system event queue attributing the
```

```
    // event to the Replot button.
    replotButton.addActionListener(
      new ActionListener(){
        public void actionPerformed(
                                    ActionEvent e){
          //Pass a 3D array of pixel data to the
          // processing object and get a modified
          // 3D array of pixel data back.  The
          // creation of the 3D array of pixel
          // data is explained later.
          threeDPixMod =
                imageProcessingObject.processImg(
                      threeDPix,imgRows,imgCols);
          //Convert the modified pixel data to a
          // 1D array of pixel data.  The 1D
          // array is explained later.
          oneDPix = convertToOneDim(
                   threeDPixMod,imgCols,imgRows);
          //Use the createImage() method to
          // create a new image from the 1D array
          // of pixel data.
          modImg = createImage(
             new MemoryImageSource(
             imgCols,imgRows,oneDPix,0,imgCols));
          //Repaint the image display frame with
          // the original image at the top and
          // the modified pixel data at the
          // bottom.
          display.repaint();
        }//end actionPerformed
      }//end ActionListener
    );//end addActionListener
    //End anonymous inner class.
    //=======================================//

    //Create a 1D array object to receive the
    // pixel representation of the image
    oneDPix = new int[imgCols * imgRows];

    //Convert the rawImg to numeric pixel
    // representation.  Note that grapPixels()
    // throws InterruptedException
    try{
      //Instantiate a PixelGrabber object
      // specifying oneDPix as the array in which
      // to put the numeric pixel data. See Sun
      // docs for parameters
      PixelGrabber pgObj = new PixelGrabber(
                      rawImg,0,0,imgCols,imgRows,
                                 oneDPix,0,imgCols);
      //Invoke the grabPixels() method on the
      // PixelGrabber object to extract the pixel
      // data from the image into an array of
      // numeric pixel data stored in oneDPix.
      // Also test for success in the process.
      if(pgObj.grabPixels() &&
```

```
                        ((pgObj.getStatus() &
                        ImageObserver.ALLBITS)
                                        != 0)){

      //Convert the pixel byte data in the 1D
      // array to int data in a 3D array to
      // make it easier to work with the pixel
      // data later.  Recall that pixel data is
      // unsigned byte data and Java does not
      // support unsigned arithmetic.
      // Performing unsigned arithmetic on byte
      // data is particularly cumbersome.
      threeDPix = convertToThreeDim(
                     oneDPix,imgCols,imgRows);

      //Instantiate a new object of the image
      // processing class.  Note that this
      // object is instantiated using the
      // newInstance method of the class named
      // Class.  This approach does not support
      // the use of a parameterized
      // constructor.
      try{
        imageProcessingObject = (
             ImgIntfc02)Class.forName(
             theProcessingClass).newInstance();

         //Post counterfeit ActionEvent to the
         // system event queue and attribute it
         // to the Replot button.  (See the
         // anonymous ActionListener class
         // defined above that registers an
         // ActionListener object on the RePlot
         // button.)  Posting this event causes
         // the image-processing method to be
         // invoked, passing the 3D array of
         // pixel data to the method, and
         // receiving a 3D array of modified
         // pixel data back from the method.
        Toolkit.getDefaultToolkit().
              getSystemEventQueue().postEvent(
                new ActionEvent(
                  replotButton,
                  ActionEvent.ACTION_PERFORMED,
                  "Replot"));

        //At this point, the image has been
        // processed and both the original
        // image and the modified image
        // have been displayed.  From this
        // point forward, each time the user
        // clicks the Replot button, the image
        // will be processed again and the new
        // modified image will be displayed
        // along with the original image.
```

```
        }catch(Exception e){
          System.out.println(e);
        }//end catch

      }//end if statement on grabPixels
      else System.out.println(
                    "Pixel grab not successful");
    }catch(InterruptedException e){
      e.printStackTrace();
    }//end catch

    //Cause the composite of the frame, the
    // canvas, and the button to become visible.
    this.setVisible(true);
    //=======================================//

    //Anonymous inner class listener to terminate
    // program.
    this.addWindowListener(
      new WindowAdapter(){
        public void windowClosing(WindowEvent e){
          System.exit(0);//terminate the program
        }//end windowClosing()
      }//end WindowAdapter
    );//end addWindowListener
    //=======================================//

  }//end constructor
  //=========================================//

  //Inner class for canvas object on which to
  // display the two images.
  class Display extends Canvas{
    //Override the paint method to display both
    // the rawImg and the modImg on the same
    // Canvas object, separated by one row of
    // pixels in the background color.
    public void paint(Graphics g){
      //First confirm that the image has been
      // completely loaded and neither image
      // reference is null.
      if (tracker.statusID(1, false) ==
                          MediaTracker.COMPLETE){
        if((rawImg != null) &&
                               (modImg != null)){
          g.drawImage(rawImg,0,0,this);
          g.drawImage(modImg,0,imgRows + 1,this);
        }//end if
      }//end if
    }//end paint()
  }//end class myCanvas
//=========================================//

  //Save pixel values as type int to make
  // arithmetic easier later.
```

```
//The purpose of this method is to convert the
// data in the int oneDPix array into a 3D
// array of ints.
//The dimensions of the 3D array are row,
// col, and color in that order.
//Row and col correspond to the rows and
// columns of the image.  Color corresponds to
// transparency and color information at the
// following index levels in the third
// dimension:
// 0 alpha
// 1 red
// 2 green
// 3 blue
// The structure of this code is determined by
// the way that the pixel data is formatted
// into the 1D array of ints produced by the
// grabPixels method of the PixelGrabber
// object.
int[][][] convertToThreeDim(
        int[] oneDPix,int imgCols,int imgRows){
  //Create the new 3D array to be populated
  // with color data.
  int[][][] data =
                  new int[imgRows][imgCols][4];

  for(int row = 0;row < imgRows;row++){
    //Extract a row of pixel data into a
    // temporary array of ints
    int[] aRow = new int[imgCols];
    for(int col = 0; col < imgCols;col++){
      int element = row * imgCols + col;
      aRow[col] = oneDPix[element];
    }//end for loop on col

    //Move the data into the 3D array.  Note
    // the use of bitwise AND and bitwise right
    // shift operations to mask all but the
    // correct set of eight bits.
    for(int col = 0;col < imgCols;col++){
      //Alpha data
      data[row][col][0] = (aRow[col] >> 24)
                                        & 0xFF;
      //Red data
      data[row][col][1] = (aRow[col] >> 16)
                                        & 0xFF;
      //Green data
      data[row][col][2] = (aRow[col] >> 8)
                                        & 0xFF;
      //Blue data
      data[row][col][3] = (aRow[col])
                                        & 0xFF;
    }//end for loop on col
  }//end for loop on row
  return data;
}//end convertToThreeDim
```

```
  //---------------------------------------//

  //The purpose of this method is to convert the
  // data in the 3D array of ints back into the
  // 1d array of type int.  This is the reverse
  // of the method named convertToThreeDim.
  int[] convertToOneDim(
         int[][][] data,int imgCols,int imgRows){
    //Create the 1D array of type int to be
    // populated with pixel data, one int value
    // per pixel, with four color and alpha bytes
    // per int value.
    int[] oneDPix = new int[
                          imgCols * imgRows * 4];

    //Move the data into the 1D array.  Note the
    // use of the bitwise OR operator and the
    // bitwise left-shift operators to put the
    // four 8-bit bytes into each int.
    for(int row = 0,cnt = 0;row < imgRows;row++){
      for(int col = 0;col < imgCols;col++){
        oneDPix[cnt] = ((data[row][col][0] << 24)
                                  & 0xFF000000)
                     | ((data[row][col][1] << 16)
                                  & 0x00FF0000)
                      | ((data[row][col][2] << 8)
                                  & 0x0000FF00)
                         | ((data[row][col][3])
                                  & 0x000000FF);
        cnt++;
      }//end for loop on col

    }//end for loop on row

    return oneDPix;
  }//end convertToOneDim
}//end ImgMod02.java class
//=========================================//

//The ProgramTest class

//The purpose of this class is to provide a
// simple example of an image-processing class
// that is compatible with the program named
// ImgMod02.

//The constructor for the class displays a small
// frame on the screen with a single textfield.
// The purpose of the text field is to allow the
// user to enter a value that represents the
// slope of a line.  In operation, the user
// types a value into the text field and then
// clicks the Replot button on the main image
// display frame.  The user is not required to
// press the Enter key after typing the new
// value, but it doesn't do any harm to do so.
```

```
//The method named processImage receives a 3D
// array containing alpha, red, green, and blue
// values for an image.  The values are received
// as type int (not type byte).

// The threeDPix array that is received is
// modified to cause a white diagonal line to be
// drawn down and to the right from the upper
// left-most corner of the image.  The slope of
// the line is controlled by the value that is
// typed into the text field.  Initially, this
// value is 1.0.  The image is not modified in
// any other way.

//To cause a new line to be drawn, type a slope
// value into the text field and click the Replot
// button at the bottom of the image display
// frame.

//This class extends Frame.  However, a
// compatible class is not required to extend the
// Frame class. This example extends Frame
// because it provides a GUI for user data input.

//A compatible class is required to implement the
// interface named ImgIntfc02.

class ProgramTest extends Frame
                          implements ImgIntfc02{

  double slope;//Controls the slope of the line
  String inputData;//Obtained via the TextField
  TextField inputField;//Reference to TextField

  //Constructor must take no parameters
  ProgramTest(){
    //Create and display the user-input GUI.
    setLayout(new FlowLayout());

    Label instructions = new Label(
                "Type a slope value and Replot.");
    add(instructions);

    inputField = new TextField("1.0",5);
    add(inputField);

    setTitle("Copyright 2004, Baldwin");
    setBounds(400,0,200,100);
    setVisible(true);
  }//end constructor

  //The following method must be defined to
  // implement the ImgIntfc02 interface.
  public int[][][] processImg(
                             int[][][] threeDPix,
```

```
                              int imgRows,
                              int imgCols){

    //Display some interesting information
    System.out.println("Program test");
    System.out.println("Width = " + imgCols);
    System.out.println("Height = " + imgRows);

    //Make a working copy of the 3D array to
    // avoid making permanent changes to the
    // image data.
    int[][][] temp3D =
                    new int[imgRows][imgCols][4];
    for(int row = 0;row < imgRows;row++){
      for(int col = 0;col < imgCols;col++){
        temp3D[row][col][0] =
                            threeDPix[row][col][0];
        temp3D[row][col][1] =
                            threeDPix[row][col][1];
        temp3D[row][col][2] =
                            threeDPix[row][col][2];
        temp3D[row][col][3] =
                            threeDPix[row][col][3];
      }//end inner loop
    }//end outer loop

    //Get slope value from the TextField
    slope = Double.parseDouble(
                             inputField.getText());

    //Draw a white diagonal line on the image
    for(int col = 0;col < imgCols;col++){
      int row = (int)(slope*col);
      if(row > imgRows -1)break;
      //Set values for alpha, red, green, and
      // blue colors.
      temp3D[row][col][0] = (byte)0xff;
      temp3D[row][col][1] = (byte)0xff;
      temp3D[row][col][2] = (byte)0xff;
      temp3D[row][col][3] = (byte)0xff;
    }//end for loop
    //Return the modified array of image data.
    return temp3D;
  }//end processImg
}//end class ProgramTest
```

Listing 31

```
/*File ImgIntfc02.java.java
Copyright 2004, R.G.Baldwin

The purpose of this interface is to declare
```

```
the one method required by image-processing
classes that are compatible with the program
named ImgMod02.java.

Tested using SDK 1.4.2 under WinXP
============================================*/

interface ImgIntfc02{
  int[][][] processImg(int[][][] threeDPix,
                       int imgRows,
                       int imgCols);
}//end ImgIntfc02
```

Listing 32

About the author

Richard Baldwin *is a college professor (at Austin Community College in Austin, TX) and private consultant whose primary focus is a combination of Java, C#, and XML. In addition to the many platform and/or language independent benefits of Java and C# applications, he believes that a combination of Java, C#, and XML will become the primary driving force in the delivery of structured information on the Web.*

Richard has participated in numerous consulting projects, and he frequently provides onsite training at the high-tech companies located in and around Austin, Texas. He is the author of Baldwin's Programming Tutorials, which has gained a worldwide following among experienced and aspiring programmers. He has also published articles in JavaPro magazine.

In addition to his programming expertise, Richard has many years of practical experience in Digital Signal Processing (DSP). His first job after he earned his Bachelor's degree was doing DSP in the Seismic Research Department of Texas Instruments. (TI is still a world leader in DSP.) In the following years, he applied his programming and DSP expertise to other interesting areas including sonar and underwater acoustics.

Richard holds an MSEE degree from Southern Methodist University and has many years of experience in the application of computer technology to real-world problems.

Baldwin@DickBaldwin.com

-end-