Processing Image Pixels using Java, Controlling Contrast and Brightness by Modifying the Color Distribution

Baldwin shows you how to control the contrast and brightness of an image by controlling the distribution of the color pixels. He provides and explains a Java program that can be used to experiment interactively with image contrast and brightness.

Published: November 30, 2004
By Richard G. Baldwin

Java Programming, Notes # 404

Preface

Third in a series

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

The first lesson in the series was entitled Processing Image Pixels using Java, Getting Started. The previous lesson was entitled Processing Image Pixels using Java, Creating a Spotlight. This lesson builds upon those earlier lessons. You will need to understand the code in the lesson entitled Processing Image Pixels using Java, Getting Started before the code in this lesson will make much sense.

Not a lesson on JAI

The lessons in this series do not provide instructions on how to use the Java Advanced Imaging *(JAI)* API. *(That will be the primary topic for a future series of lessons.)* The purpose of this series is to teach you how to implement common image-processing algorithms by working directly with the pixels.

A framework or driver program

The lesson entitled Processing Image Pixels using Java, Getting Started provided and explained a program named **ImgMod02** that makes it easy to:

- Manipulate and modify the pixels that belong to an image.
- Display the modified image along with the original image.

ImgMod02 serves as a driver that controls the execution of a second program that actually processes the pixels.

> *(See the comments regarding an updated version of **ImgMod02** later in this lesson.)*

The program that I will explain in this lesson runs under the control of **ImgMod02**. You will need to go to the lesson entitled Processing Image Pixels using Java, Getting Started and get copies of the program named **ImgMod02** and the interface named **ImgIntfc02** in order to compile and run the program that I will provide in this lesson.

Controlling image contrast and brightness

The image-processing program that I will explain in this lesson will show you how to control image ***contrast*** and ***brightness*** by controlling the distribution of color values that make up the image.

> *(This usage of the word **contrast** originated in the days of black and white TV and grayscale images. I'm not sure that it is an entirely appropriate word for describing the relationship between the distribution of color values in an image and the appearance of that image. However, I'm not aware of another word that is used for this purpose, so I will stick with the word contrast.)*

Future lessons will show you how to create a number of other special effects by directly modifying the pixels belonging to an image.

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.

Controlling the contrast and the brightness of an image

Figure 1 shows a *before* and *after* view of a digital photograph that I took somewhere while on vacation. The image at the top was cropped out of the original photograph. The image at the bottom shows the result of increasing the contrast and brightness by modifying the distribution of the color values.

Figure 1 Before and after image processing.

As you can see, there is a marked difference between the appearances of the two images. The colors in the processed image at the bottom are much more vibrant than the colors in the original image at the top. In addition, the processed image shows much more detail than the original image.

In summary, the processed image is brighter and has more contrast.

User input and color distribution

Figure 2 shows the user input values at the top that were used to produce the output shown in Figure 1.

Figure 2 also shows the histograms of the color values contained in the two images in Figure 1 *before* and *after* processing. The narrow histogram in the green area in the middle is the

histogram of the color values in the original image. The broader histogram in the orange area at the bottom is the histogram of the processed image.

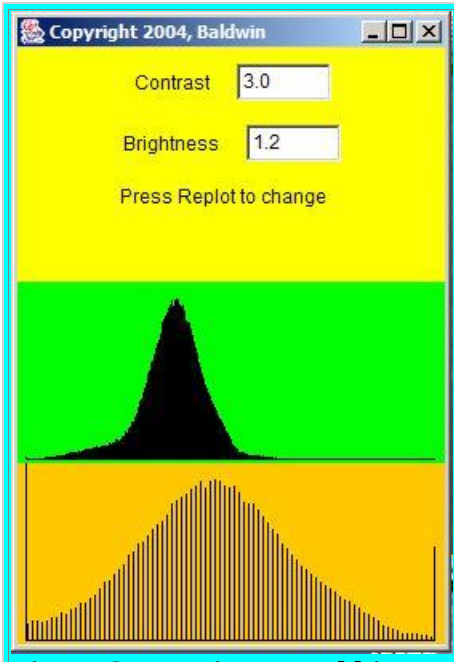

Figure 2 User input and histograms.

Increased distribution width and mean value

In this case, the width of the histogram *(contrast)* was increased by a factor of 3.0 as a result of the user input shown at the top of Figure 2. In addition, the mean value of the histogram was increased by a factor of 1.2 as a result of user input.

The increase in the mean value of the distribution caused the processed image to be brighter. The increase in the width of the distribution caused the image to contain a wider range of colors, thus improving the contrast.

Numeric statistical values

Figure 3 shows the mean value and the rms value in numeric terms before and after processing.

```
Width = 294
Height = 321

Original mean: 91
Original rms: 22
New rms: 66
New mean: 111
```

Figure 3 *Before* and *after* statistics.

The information shown in Figure 3 is displayed on the command-line screen when the program is run. The numeric results in Figure 3 are consistent with the user input values shown in Figure 2.

Display format

The output shown in Figure 1 was produced by the driver program named **ImgMod02**. The output shown in Figure 2 and Figure 3 was produced by the image-processing program named **ImgMod23**.

As in all of the graphic output produced by the driver program named **ImgMod02**, the original image is shown at the top and the modified image is shown at the bottom.

An interactive image-processing program

The image-processing program illustrated by Figures 1, 2, and 3 allows the user to interactively control the contrast and brightness of the modified image. Depending on user input, the brightness can range from very light to very dark, and the contrast can range from maximum contrast at one extreme to being totally washed out at the other extreme.

Theoretical basis and practical implementation

While discussing the lessons in this series, I will provide some of the theoretical basis for special-effects algorithms. In addition, I will show you how to implement those algorithms in Java.

Background Information

The earlier lesson entitled Processing Image Pixels using Java, Getting Started provided a great deal of background information as to how images are constructed, stored, transported, and rendered. I won't repeat that material here, but will simply refer you to the earlier lesson.

The earlier lesson introduced and explained the concept of a pixel. In addition, the lesson provided a brief discussion of image files, and indicated that the program named **ImgMod02** is compatible with ***gif*** files, ***jpg*** files, and possibly some other file formats as well.

The lessons in this series are not particularly concerned with file formats. Rather, the lessons are concerned with what to do with the pixels after they have been extracted from an image file.

> *(However, this lesson will demonstrate that a great deal more color information is contained in a jpg file than is contained in a gif file for the same image.)*

A three-dimensional array of pixel data as type int

The driver program named **ImgMod02**:

- Extracts the pixels from an image file.
- Converts the pixel data to type **int.**
- Stores the pixel data in a three-dimensional array of type **int** that is well suited for processing.
- Passes the three-dimensional array object's reference to an image-processing program.
- Receives back a reference to a three-dimensional array object containing modified pixel data.
- Displays the original image and the modified image in a stacked display as shown in Figure 1.
- Makes it possible for the user to provide new input data to the image-processing program, invoke the image-processing program again, and create a new display showing the newly-modified image along with the original image.

The manner in which that is accomplished was explained in the earlier lesson entitled Processing Image Pixels using Java, Getting Started.

Will concentrate on the three-dimensional array of type int

This and future lessons in this series will show you how to write image-processing programs that implement a variety of image-processing algorithms. The image-processing programs will receive raw pixel data in the form of a three-dimensional array of type **int**, and will return modified pixel data in the form of a three-dimensional array of type **int**.

A grid of colored pixels

Each three-dimensional array object represents one image consisting of a grid of colored pixels. The pixels in the grid are arranged in rows and columns when they are rendered. One of the dimensions of the array represents rows. A second dimension represents columns. The third dimension represents the color *(and transparency)* of the pixels.

Fundamentals

Once again, I will refer you to the earlier lesson entitled Processing Image Pixels using Java, Getting Started to learn:

- How the primary colors of red, green, and blue and the transparency of a pixel are represented by four ***unsigned*** 8-bit bytes of data.
- How specific colors are created by mixing different amount of red, green, and blue.
- How the range of each primary color and the range of transparency extends from 0 to 255.
- How black, white, and the colors in between are created.
- How the overall color of each individual pixel is determined by the values stored in the three color bytes for that pixel, as modified by the transparency byte.

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

The program that I will discuss in this lesson requires the program named **ImgMod02** and the interface named **ImgIntfc02** for compilation and execution. I provided and explained that material in the earlier lesson entitled Processing Image Pixels using Java, Getting Started.

I will present and explain a new Java program named **ImgMod23** in this lesson. This program, when run under control of the program named **ImgMod02**, will produce outputs similar to Figures 1, 2, and 3.

> *(The results will be different if you use a different image file or provide different user input values.)*

The processImg method

The program named **ImgMod23**, *(and all image-processing programs that are capable of being driven by **ImgMod02**)*, must implement the interface named **ImgIntfc02**. That 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 named **ImgMod02** causes the original image and the modified image to be displayed in a frame with the original image above the modified image *(see Figure 1 for an example of the display format).*

Usage information for ImgMod02 and ImgMod23

To use the program named **ImgMod02** to drive the program named **ImgMod23**, enter the following at the command line:

```
java ImgMod02 ImgMod23 ImagePathAndFileName
```

The image file

The image file can be a ***gif*** file or a ***jpg*** file. Other file types may be compatible as well. If the program is unable to load the image file within ten seconds, it will abort with an error message.

> *(You should be able to right-click on the image in Figure 6 to download and save the image locally. Then you should be able to replicate the output produced in Figures 1, 2, and 3.)*

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. The two images are identical when the program first starts running.

A **Replot** button appears at the bottom of the frame. If the user clicks the **Replot** button, the **processImg** 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 named **ImgMod23** provides a GUI for user input, as shown in Figure 2. This makes it possible for the user to provide different values for Contrast and Brightness each time the image-processing method is rerun. To rerun the image-processing method, type the new values into the text fields and press the **Replot** button.

Discussion and Sample Code

PixelGrabber versus WritableRaster

After publishing the lesson entitled Processing Image Pixels using Java, Getting Started and after putting the lesson entitled Processing Image Pixels using Java, Creating a Spotlight into the publication pipeline, I found this review of the first lesson at http://community.java.net/javadesktop/.

> *"... the 2D team doesn't like to see PixelGrabber in post-1.1 code. You can use BufferedImage.getRaster() and Raster.getDataBuffer() instead."*

Processing pixels versus getting pixels

The primary purpose of this series of lessons is to teach you about pixel-processing algorithms as opposed to teaching you how to gain access to the pixels. However, assuming that the *2D team* has a good reason for the above-stated preference, I decided to provide you with a version of the program that uses *BufferedImage.getRaster()* instead of *PixelGrabber*. That version of the program is provided in Listing 21 near the end of the lesson.

An updated driver program

The program named **ImgMod02a** in Listing 21 is an updated version of the program named **ImgMod02** provided in the lesson entitled Processing Image Pixels using Java, Getting Started. You can easily find the changes in the new version of the program by searching the source code for the name of the class **BufferedImage**.

Both programs do the same thing, so you can use whichever one you prefer to run the programs in this and subsequent lessons.

The processImg method

The image-processing program must implement the interface named **ImgIntfc02**. A listing of that interface was provided in the earlier lesson entitled Processing Image Pixels using Java, Getting Started. 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 an incoming three-dimensional array of pixel data stored as type **int**. The second and third parameters specify the number of rows and the number of columns of pixels in the image.

It's best to make and modify a copy

Normally the **processImg** method 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.

Be careful of the range of values

The **processImg** method is free to modify the values of the pixels in any manner whatsoever before returning the modified array. Note however that native pixel data consists of four *unsigned* bytes, whereas the **processImg** method works with pixel values of the *signed* type **int**.

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 what you expect to see.

Dealing with out-of-range values

There are a variety of ways to deal with out-of-range values. The program that I will explain in this lesson simply clips out-of-range values at 0 and 255. Future lessons will deal with out-of-range values in other ways.

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

Instantiate an image-processing object

During execution, the program named **ImgMod02** reaches a point where it has captured the pixel data from the original image file into a three-dimensional array of type **int** suitable for processing. Then it invokes the **newInstance** method of the class named **Class** to instantiate an object of the image-processing class.

Invoke the processImg method

At this point, the program named **ImgMod02**:

- Has the pixel data in the correct format
- Has an image-processing object that will process those pixels and will return an array containing modified pixel values

All that the **ImgMod02** program needs to do at this point is to invoke the **processImg** method on the image-processing object passing the pixel data along with the number of rows and columns of pixels as parameters.

Posting a counterfeit ActionEvent

The **ImgMod02** program 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 pressed the **Replot** button shown in Figure 1.

In either case, the **actionPerformed** method is invoked on an **ActionListener** object that is registered on the **Replot** button. The code in the **actionPerformed** method invokes the **processImg** method on the image-processing object.

The three-dimensional array of pixel data is passed to the **processImg** method. The **processImg** method returns a three-dimensional array of modified pixel data, which is displayed as an image below the original image as shown in Figure 1.

The program named ImgMod23

The purpose of this program is to show you how to modify the contrast and the brightness of an image by modifying the distribution of the color values.

The contrast

The contrast of an image is determined by the width of the distribution of the color values belonging to the image. If all color values are grouped together in a narrow distribution *(as in the top distribution in Figure 2)*, the details in the image will tend to be washed out. In addition,

the overall appearance of the image may tend toward some shade of gray. The shade of gray will depend on the location of the grouping between the extremes of 0 and 255.

> *(If every color value in the image is the same, the distribution will have a single peak and the entire image will be black, white, or some shade of gray. This is the ultimate in narrow distributions.)*

Location of the grouping of color values

If the color values are closely grouped near zero, the colors will range from black to dark gray. If the color values are closely grouped near 255, the colors will range from very light gray to white.

Controlling the contrast

The contrast of an image can be increased by increasing the width of the distribution of the color values, as was done for the bottom image in Figure 1. Figure 2 shows the distribution of color values before and after processing.

> *(Figure 3 shows that the width of the distribution of the bottom image was increased by a factor of 3.0 relative to the width of the distribution for the top image. The rms value of the distribution increased from 22 to 66.)*

The contrast can be decreased by decreasing the width of the distribution.

Controlling the brightness

The overall brightness of an image is determined by the location of the grouping of color values. If the location tends to be near the upper limit of 255, the image will tend to be very bright. If the location tends to be near the lower limit of 0, the image will tend to be very dark.

> *(The distribution for the original image in Figure 1 was to the left of center as shown in Figure 2. The mean value of the original distribution was 91 with the center point being 128.)*

The brightness of an image can be increased by moving the grouping toward 255, as was done for the bottom image in Figure 1.

> *(Figure 2 shows that the mean value of the distribution for the bottom image was moved to the right by a factor of 1.2. As a result, the mean value of the processed image was at 111, instead of 91, with the center being at 128.)*

The overall brightness can be decreased by moving the grouping toward 0.

Changing the location of the distribution

A straightforward way to change the location of the distribution is to add *(or subtract)* the same value to or from every color value. This will change the brightness without changing the contrast.

Changing the width of the distribution

A straightforward way to change the width of the distribution, with or without changing the general location of the distribution, consists of the following steps:

- Calculate the mean or average value of all the color values.
- Subtract the mean from every color value, causing the distribution to be shifted to the left, with a new mean value of zero.
- Multiply every color value by the same scale factor. If the scale factor is greater than 1.0, the width of the distribution will be increased. If the scale factor is less than 1.0, the width of the distribution will be decreased.
- Add the original mean value *(or some other value)* to every color value to either restore the distribution to its original location or to move it to a different location. *(This program allows the user to add a new mean value that is different from the original mean value in order to change the brightness of the image.)*

The above steps will modify the width of the distribution and change the contrast of the image. If the original mean value is added in the final step, the brightness won't be changed. If some other value is added in the final step, the brightness will also be changed.

Protecting against out-of-range values

After performing the operations listed above, it is important to make certain that all color values fall within the range of an *unsigned* byte. This requires eliminating all color values that are greater than 255 and all color values that are less than 0. A simple way to do this is to clip the color values at those two limits if they fall outside the limits.

Clipping can change the distribution

Remember, however, that clipping can change the width, the shape, and the mean value of the distribution. Clipping values to the limits will tend to narrow the distribution and to create a spike in the distribution at the value of the limit.

> *(Note the two spikes at the ends of the distribution at the bottom of Figure 2. These spikes are the result of clipping the modified color values at 0 and 255.)*

Measuring the width of a distribution

There are several ways to measure the width of a distribution. One way is to measure the distance between the minimum value and the maximum value. This is not a very good way however, because a couple of outliers can lead to erroneous conclusions regarding the width of the distribution.

The root mean square *(rms)* value

A better way is to compute the root mean square *(rms)* value of all the color values in the distribution. This approach is less sensitive to outliers and produces an estimate of the distribution width that is more representative of the bulk of the distribution.

For certain known distribution shapes ...

For distributions having certain known shapes, it is possible to predict with some accuracy the percentage of the color values that will fall within a range bounded by the mean value plus and minus the rms value.

> *(For example, if the distribution of color values is uniform, which it probably won't be, approximately 60-percent of the color values will probably fall within a range bounded by the mean value plus and minus the rms value. In other words, approximately 60-percent of the color values will probably fall within a range equal to twice the rms value located near the center of the distribution.)*

For unknown distribution shapes ...

If the shape of the distribution is not known, we can say that for an image containing a large number of different color values, *(as is the case for the photographic images in Figure 1),* a large percentage of those color values will lie within the range bounded by the mean value plus and minus the rms value.

Computing and displaying the rms value

The rms value is computed and displayed in this program solely to provide information to the user. The rms value is not used in the computations that control the contrast and brightness of the image.

For most images containing a large number of colors, the user should be able to see a direct correlation between the rms value and the contrast. For small rms values, the contrast in the image will probably appear to be washed out and the overall color of the image may tend towards gray.

The Graphical User Interface

The program provides a GUI with two text fields and two histograms as shown in Figure 2.

The text fields make it possible for the user to modify the contrast and brightness of the processed image by entering new values for Contrast and Brightness and then pressing the **Replot** button on the main display.

The input values are multiplicative factors

The values entered into the text fields are multiplicative factors. The initial value in each text field is 1.0. For these values, the processed image should be identical to the original image.

To increase the contrast or the brightness, type a value greater than 1.0 into the corresponding text field and press the **Replot** button.

To decrease the contrast or the brightness, type a value less than 1.0 into the corresponding text field and press the **Replot** button.

> *(For most images containing a large number of color values, the new distribution width will approximate the old distribution width multiplied by the value in the Contrast field. Similarly, the new mean value of the distribution will approximate the old mean value multiplied by the value in the Brightness field. Be aware, however, that clipping at 0 and 255 can introduce errors into these estimates.)*

No need to press the Enter key

It isn't necessary to press the **Enter** key to type the new values into the text fields, but doing so won't cause any harm.

Entering a text string that cannot be converted to a value of type **double** will cause the program to throw an exception.

Normalized histograms

The top histogram in Figure 2 shows the distribution of color values for the original image. The bottom histogram shows the distribution of color values for the modified image.

The histogram values are normalized to a peak value of 100, exclusive of the extreme values at 0 and 255. The two values at the extremes can, and often do, exceed 100 and will often go out of plotting range on the histograms.

Transparent areas

Note that the pixel modification in this program has no impact on transparent pixels. If you don't see what you expect to see when you run the program, it may be because your image contains large transparent areas.

Testing

The program was tested using SDK 1.4.2 and WinXP

Will discuss in fragments

I will break the program down into fragments for discussion. A complete listing of the program is provided in Listing 20 near the end of the lesson.

The class definition for the program named **ImgMod23** begins in Listing 1.

```
class ImgMod23 extends Frame
                            implements
ImgIntfc02{

  TextField contrastField;
  TextField brightField;
  Panel input;
  OrigHistogramPanel origHistPanel;
  NewHistogramPanel newHistPanel;
  int[] origHistogram = new int[256];
  int[] newHistogram = new int[256];
```

Listing 1

The class extends **Frame**, because an object of the class is the GUI shown in Figure 2.

The class implements **ImgIntfc02** to make it compatible with the driver program named **ImgMod02**.

Listing 1 declares several variables that will be used later, and initializes some of them.

The constructor

The entire constructor is shown in Listing 2. The whole purpose of the constructor is to create the GUI shown in Figure 2.

```
  ImgMod23(){
    //Create a Box container with a
vertical
    // layout and place it in the center
of
    // the Frame.
    Box aBox = new Box(BoxLayout.Y_AXIS);
    this.add(aBox,BorderLayout.CENTER);

    //Create and place the user input
panel at
    // the top of the vertical stack.
Make it
    // yellow
    input = new Panel();

    //Create panels to group the labels
with the
    // text fields and add them to the
input
    // panel under FlowLayout.
    Panel contrastPanel = new Panel();
```

```
    contrastPanel.add(new
Label("Contrast"));
    contrastField = new
TextField("1.0",5);
    contrastPanel.add(contrastField);
    input.add(contrastPanel);

    Panel brightnessPanel = new Panel();
    brightnessPanel.add(new
Label("Brightness"));
    brightField = new TextField("1.0",5);
    brightnessPanel.add(brightField);
    input.add(brightnessPanel);

    input.add(new Label(
                      "Press Replot to
change"));

    input.setBackground(Color.YELLOW);
    aBox.add(input);

    //Create and place the panel for the
    // original histogram in the middle
of the
    // stack.  Make it green.
    origHistPanel = new HistogramPanel();

origHistPanel.setBackground(Color.GREEN);
    aBox.add(origHistPanel);

    //Create and place the panel for the
new
    // histogram at bottom of the stack.
Make
    // it orange.
    newHistPanel = new HistogramPanel();

newHistPanel.setBackground(Color.ORANGE);
    aBox.add(newHistPanel);

    //Set miscellaneous properties.
    setTitle("Copyright 2004, Baldwin");
    setBounds(400,0,275,400);
    setVisible(true);
  }//end constructor
```

Listing 2

While the constructor is somewhat long and tedious, there is nothing here having much to do with processing pixels so I won't discuss it in detail. If you see material in Listing 2 that you don't understand, you will probably find explanations in the other lessons on my website.

The histogram panel

The top histogram shown in Figure 2 was plotted on an object instantiated from the inner class named **OrigHistogramPanel** that is defined in Listing 3.

The bottom histogram in Figure 2 was plotted on an object instantiated from an almost identical inner class named **NewHistogramPanel**.

You can view the class definition for **NewHistogramPanel** in Listing 20 near the end of the lesson.

```
  class OrigHistogramPanel extends
Panel{
    public void paint(Graphics g){
      //Following constant corrects
for positive
      // direction on the y-axis.
      final int flip = 110;
      //Following constant is used to
shift the
      // histogram plot 5 pixels to
the right.
      final int shift = 5;
      //Draw the horizontal axis
      g.drawLine(0 + shift,flip,
                                255 +
shift,flip);
      //Draw the histogram
      for(int cnt = 0;cnt <
origHistogram.length;

cnt++){
        g.drawLine(cnt + shift,flip -
0,
                   cnt + shift,
                     flip -
origHistogram[cnt]);
      }//end for loop
    }//end paint
  }//end class OrigHistogramPanel
```

Listing 3

The purpose of defining the **OrigHistogramPanel** class and the **NewHistogramPanel** class was to make it possible to override the **paint** method to cause it to draw histograms.

I have already explained all of the concepts embodied in Listing 3 in numerous previous lessons on plotting. Therefore, the code in Listing 3 shouldn't require a detailed discussion.

The processImg method

The **processImg** method begins in Listing 4. This is where the code begins to get a little more interesting.

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

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

    //Get user input values for
contrast and
    // brightness.  These values will
be used as
    // multipliers to change the
contrast and
    // the brightness.
    double contrast =
Double.parseDouble(

contrastField.getText());
    double brightness =
Double.parseDouble(

brightField.getText());
```

Listing 4

The **processImg** method receives a three-dimensional array of type **int** containing pixel data. The method also receives the number of rows of pixels and the number of columns of pixels in the image described by the data in the three-dimensional array.

The method begins by printing the width and the height of the image in pixels as information to the user.

Then the method gets the two text strings from the text fields in Figure 2 and converts them to values of type **double**.

Make a working copy of the array

Next, the **processImg** method makes a working copy of the three-dimensional array containing pixel data to avoid making any permanent changes to the original pixel data. This is shown in Figure 5.

```
    int[][][] output3D =
                      new
int[imgRows][imgCols][4];
```

```
    for(int row = 0;row <
imgRows;row++){
      for(int col = 0;col <
imgCols;col++){
        output3D[row][col][0] =

threeDPix[row][col][0];
        output3D[row][col][1] =

threeDPix[row][col][1];
        output3D[row][col][2] =

threeDPix[row][col][2];
        output3D[row][col][3] =

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

Listing 5

A reference to the working copy of the three-dimensional array is stored in the reference variable named **output3D**. This array contains the pixel data that will be modified and returned.

Remove the mean value

The code in Listing 6 gets, saves, displays, and removes the mean value from the color values in the working array.

```
    int mean =
getMean(output3D,imgRows,imgCols);
    System.out.println("Original mean: " +
mean);


removeMean(output3D,imgRows,imgCols,mean);
```

Listing 6

Listing 6 begins by invoking the method named **getMean** to get the mean value of all the color values in the image. It displays this value on the command-line screen for the benefit of the user.

Then Listing 6 invokes the method named **removeMean** to cause the distribution to be shifted to the left so that it will have a mean value of zero. Note that the mean value is passed to the **removeMean** method.

The getMean method

The **getMean** method is shown in its entirety in Listing 7

```
  int getMean(int[][][] data3D,int
imgRows,

int imgCols){
    int pixelCntr = 0;
    long accum = 0;
    for(int row = 0;row <
imgRows;row++){
      for(int col = 0;col <
imgCols;col++){
        accum += data3D[row][col][1];
        accum += data3D[row][col][2];
        accum += data3D[row][col][3];
        pixelCntr += 3;
      }//end inner for loop
    }//end outer for loop

    return (int)(accum/pixelCntr);

  }//end getMean
```

Listing 7

The code in the **getMean** method is straightforward and shouldn't require further explanation.

The removeMean method

The **removeMean** method is shown in Listing 8.

```
  void removeMean(int[][][] data3D,int
imgRows,
                          int
imgCols,int mean){
    for(int row = 0;row <
imgRows;row++){
      for(int col = 0;col <
imgCols;col++){
        data3D[row][col][1] -= mean;
        data3D[row][col][2] -= mean;
        data3D[row][col][3] -= mean;
      }//end inner for loop
    }//end outer for loop
  }//end removeMean
```

Listing 8

The code in the **removeMean** method is also straightforward and shouldn't require further discussion.

Get and display the rms value of the original image

Returning now to the **processImg** method, the code in Listing 9 gets and displays the rms value for the color value distribution of the original image.

> *(Although the working copy of the three-dimensional array is passed to the **getRms** method, nothing has been done to the color values at this point that would change the rms value.)*

```
    int rms =
getRms(output3D,imgRows,imgCols);
    System.out.println("Original rms:
" + rms);
```

Listing 9

The rms value is for user information only. It is not used by any of the computations made by the program for modifying contrast and brightness.

The getRms method

The **getRms** method is shown in Listing 10.

```
  int getRms(int[][][] data3D,int
imgRows,

int imgCols){
    int pixelCntr = 0;
    long accum = 0;
    for(int row = 0;row <
imgRows;row++){
      for(int col = 0;col <
imgCols;col++){
        accum += data3D[row][col][1] *

data3D[row][col][1];
        accum += data3D[row][col][2] *

data3D[row][col][2];
        accum += data3D[row][col][3] *

data3D[row][col][3];
        pixelCntr += 3;
      }//end inner for loop
    }//end outer for loop
    int meanSquare =
(int)(accum/pixelCntr);
    int rms =
(int)(Math.sqrt(meanSquare));
    return rms;
  }//end getRms
```

```
Listing 10
```

For those who may not be familiar with the rms concept, if a distribution has a mean value of zero, *(which this one does have at this point),* calculation of the rms value is straightforward.

> *(Calculation of the rms value is a little more complicated if the mean value of the distribution is not zero.)*

Calculating the rms value

For this case, the rms value is equal to the square root of the average of the squared values of all the color values in the population. That is to say:

- Compute the sum of the squares of all the values.
- Divide the sum by the number of values that were added into the sum.
- Compute the square root of that quotient.

Knowing this, you shouldn't need any further explanation of the code in Listing 10.

Scale the distribution

Returning once again to the **processImg** method, Listing 11 invokes the **scale** method to cause every color value to be multiplied by the multiplicative factor extracted from the Contrast field in Figure 2.

```
scale(output3D,imgRows,imgCols,contrast);
    System.out.println("New rms: "
              +
getRms(output3D,imgRows,imgCols));

Listing 11
```

Multiplying every color value by the same multiplicative factor either widens or narrows the distribution.

> *(Note that this only works for a distribution with a mean value of zero. If the mean value is not zero, the arithmetic required to produce the same result is somewhat more complicated.)*

Then the code in Listing 11 gets and displays the rms value for the modified color values.

The scale method

The entire **scale** method is shown in Listing 12. The code in this method is straightforward.

```
  void scale(int[][][] data3D,int
imgRows,
                     int
imgCols,double scale){
    for(int row = 0;row <
imgRows;row++){
      for(int col = 0;col <
imgCols;col++){
        data3D[row][col][1] *= scale;
        data3D[row][col][2] *= scale;
        data3D[row][col][3] *= scale;
      }//end inner for loop
    }//end outer for loop
  }//end scale
```

Listing 12

Restoring to a non-zero mean value

Returning once again the **processImg** method, Listing 13 invokes the **shiftMean** method to cause the mean value of the distribution to be shifted from zero towards 255.

> *(The mean could also be shifted in a negative direction, but that probably wouldn't be very useful.)*

```
shiftMean(output3D,imgRows,imgCols,

(int)(brightness*mean));
    System.out.println("New mean: "
            +
getMean(output3D,imgRows,imgCols));
```

Listing 13

The shift in the mean value is accomplished by adding the same value to every color value in the image.

> *(Note that the value being added is the product of the mean value of the original image and the value extracted from the Brightness field in Figure 2.)*

Behavior at program startup

Thus, at startup when the value in that field is 1.0, this simply restores the mean value to the mean value of the original image. Later when the user enters a value other than 1.0, this process shifts the mean value to a value that is different from the mean value of the original image.

Then Listing 13 gets and displays the new mean value of the modified image.

The shiftMean method

Listing 14 shows the entire **shiftMean** method. The code in this method is straightforward.

```
  void shiftMean(int[][][] data3D,int
imgRows,
                         int
imgCols,int newMean){
    for(int row = 0;row <
imgRows;row++){
      for(int col = 0;col <
imgCols;col++){
        data3D[row][col][1] +=
newMean;
        data3D[row][col][2] +=
newMean;
        data3D[row][col][3] +=
newMean;
      }//end inner for loop
    }//end outer for loop
  }//end shiftMean
```

Listing 14

Clipping the color values

Listing 15 invokes the **clip** method to ensure that none of the color values in the modified image are outside the range from 0 to 255 *(the value range of an unsigned byte).*

```
    clip(output3D,imgRows,imgCols);
    System.out.println();
```

Listing 15

Then the code in Listing 15 prints a blank line to serve as a separator in the screen output between runs.

The clip method

The **clip** method is shown in Listing 16. The code in this method is straightforward.

```
  //Method to clip the color data at 0
and 255
  void clip(int[][][] data3D,int
imgRows,

int imgCols){
    for(int row = 0;row <
```

```
imgRows;row++){
      for(int col = 0;col <
imgCols;col++){
        if(data3D[row][col][1] < 0)
          data3D[row][col][1] = 0;
        if(data3D[row][col][1] > 255)
          data3D[row][col][1] = 255;

        if(data3D[row][col][2] < 0)
          data3D[row][col][2] = 0;
        if(data3D[row][col][2] > 255)
          data3D[row][col][2] = 255;

        if(data3D[row][col][3] < 0)
          data3D[row][col][3] = 0;
        if(data3D[row][col][3] > 255)
          data3D[row][col][3] = 255;

      }//end inner for loop
    }//end outer for loop
  }//end clip
```

Listing 16

The end of the process

That completes the process of modifying the contrast and/or the brightness. The modified array data could be returned for display at this point.

Let's see some histograms

Before returning, however, the **processImg** method creates and displays histograms for the distributions of the original image and the modified image. The histograms are provided for instructional purposes and have nothing to do with the process of modifying the contrast and/or the brightness of the image.

Create and display the histograms

Returning to the **processImg** method, the code in Listing 17 invokes the **getHistogram** method twice in succession to get a histogram for:

- The original image represented by the array referred to by **threeDPix**
- The modified image represented by the array referred to by **output3D**

```
    //Create and draw the two
histograms
    origHistogram =
getHistogram(threeDPix,

imgRows,imgCols);
```

```
    origHistPanel.repaint();


    newHistogram =
getHistogram(output3D,

imgRows,imgCols);
    newHistPanel.repaint();

Listing 17
```

Draw the histograms

Once each histogram has been created and returned, Listing 17 invokes the **repaint** method on the corresponding histogram panel. This causes the overridden **paint** method for the histogram panel *(see Listing 3)* to be invoked, drawing the histogram in the panel.

The getHistogram method

The **getHistogram** method creates a histogram for the color data belonging to an incoming image and returns the histogram values in an array object of type **int**. The **getHistogram** method is shown in Listing 18.

```
  int[] getHistogram(int[][][] data3D,
                     int
imgRows,int imgCols){
    //Count and record occurrences of
values
    int[] hist = new int[256];
    for(int row = 0;row <
imgRows;row++){
      for(int col = 0;col <
imgCols;col++){
        hist[data3D[row][col][1]]++;
        hist[data3D[row][col][2]]++;
        hist[data3D[row][col][3]]++;
      }//end inner for loop
    }//end outer for loop

    //Get the maximum value, exclusive
of the
    // values at 0 and 255
    int max = 0;
    for(int cnt = 1;cnt < hist.length
- 1;cnt++){
      if(hist[cnt] > max){
        max = hist[cnt];
      }//end if
    }//end for loop

    //Normalize histogram to a peak
```

```
value of 100
    // based on the max value,
exclusive of the
    // values at 0 and 255
    for(int cnt = 0;cnt <
hist.length;cnt++){
      hist[cnt] = 100 * hist[cnt]/max;
    }//end for loop
    return hist;
  }//end getHistogram
```

Listing 18

Basically the method examines all of the color values that belong to the image, counting and recording the number of occurrences of each of the values between 0 and 255 inclusive.

Normalization

The count values are then normalized to cause the largest count to have a value of 100 *(exclusive of the counts for the values of 0 and 255).*

A large image with lots of pixels can produce very large count values, while a smaller image will produce smaller count values. The normalization is performed to bring the results into a suitable plotting range regardless of the number of pixels in the image.

Excluding 0 and 255 from the normalization

The values of 0 and 255 are excluded from the normalization process. As a result of clipping, the counts of those two values can be very high.

> *(Images with large amounts of white space or black space will also produce very high count values at 255 and 0.)*

If these two values are included in the normalization process, they will often dominate the process and cause all of the other normalized count values to be so relatively small that they can't be seen in the plot.

Because these values are excluded from the normalization process, the plotted histograms will often have spikes at either end that go completely off the top of the plot.

> *(That is the case on the left end of the bottom histogram in Figure 2.)*

Now that you know what the code in Listing 18 is designed to do, understanding that code should be straightforward.

Return the modified image data

The code in Listing 19 returns a reference to the three-dimensional array object containing the modified image data, suitable for display by the program named **ImgMod02**.

```
    return output3D;

  }//end processImg
```

Listing 19

Listing 19 also signals the end of the **processImg** method, and the end of the image-processing program named **ImgMod23**.

Run the Program

I encourage you to copy, compile, and run the program named **ImgMod23** provided in this lesson. Experiment with it, making changes and observing the results of your changes.

> *(Remember, you will also have to copy the program named **ImgMod02** and the interface named **ImgIntfc02** from the earlier lesson entitled* Processing Image Pixels using Java, Getting Started*. As an alternative to **ImgMod02**, you can use the program named **ImgMod02a** provided in Listing 21 below.)*

Replicate Figures 1, 2, and 3

To replicate the output shown in Figures 1, 2, and 3, right-click and download the jpg image file in Figure 6 below. You should be able to use that file to replicate the output shown in those figures.

Experiment with the file, holding the Brightness value constant while changing the Contrast value. Observe the results of making these changes. Hold the Contrast value constant while changing the Brightness value and observe the results of those changes as well. Then change both values at the same time and observe the results.

Set the Contrast and Brightness values to the values shown in Figure 2 and confirm that your output matches my output shown in Figure 1.

Processing a gif image

Then right-click and download the gif image file in Figure 7. This file is a gif version of the same image as the jpg version in Figure 6. The jpg version from Figure 6 was used to produce the output shown in Figures 1, 2, and 3. Run the program using the gif image in an attempt to replicate the output shown in Figures 1, 2, and 3.

Color distribution of the gif image

Figure 4 shows the color distribution for the gif image under the same conditions as the color distribution for the jpg image shown in Figure 2. Although it isn't shown here, the mean values and the rms values for the two versions of the image are essentially the same.

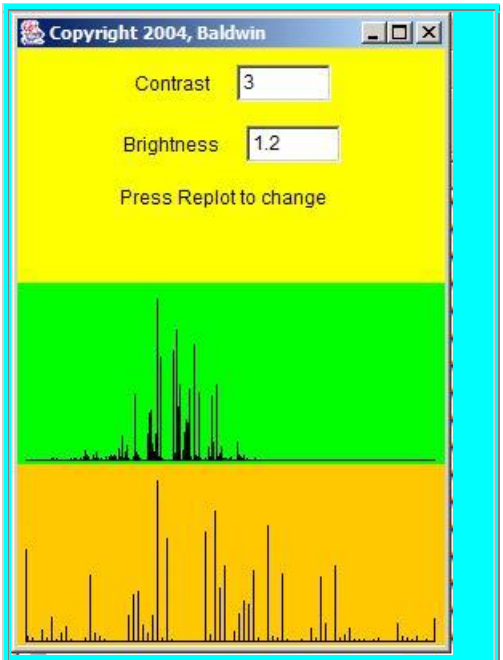

Figure 4 Distribution for gif image

Note in particular the distribution of color values for the gif image shown in Figure 4. Compare this with the distribution of color values for the jpg image shown in Figure 2.

A more sparse color distribution

Although the mean and rms values are the same between the gif and jpg versions, the color distribution for the gif image in Figure 4 is much more sparse than the color distribution for the jpg image in Figure 2.

This is because the gif file format is severely limited in terms of the number of actual colors it can preserve. In other words, a gif image contains less color information than a jpg image.

Enhance the gif image

Try to enhance the gif image from Figure 7 the same way that you enhanced the jpg image from Figure 6. Open a second command-line window and get both outputs on the screen at the same time so that you can do a side-by-side comparison.

If you look closely, you should see that the result of enhancing the jpg image is somewhat better than the result of enhancing the gif image. There is more *noise* in the enhanced gif image. This is due to the fact that the jpg image contains more color information than the gif image.

From this, you might conclude that if you are interested in image quality, you should work with jpg images instead of gif images whenever possible.

A hand-drawn image

Now experiment with an image drawn by someone using a typical paint program, such as the image in Figure 8. Unless the person who made the drawing took great pains to use lots of different colors in the image, the color distribution will probably be radically different from the color distribution of a digital photograph like Figure 6.

Figure 5 shows the color distribution for the hand-drawn image in Figure 8.

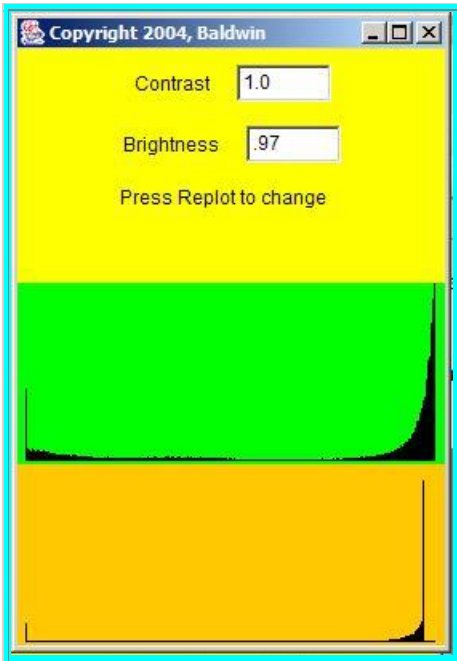

Figure 5 Color distribution of drawn image.

A very large peak at 255

As you can see, the color distribution for the original image in Figure 8 has peaks at 0 and 255 with very little in between. This is because the colors typically provided by paint programs tend to be at or near full intensity with values near 255. Also, the large white area in the image in Figure 8 requires a large number of pixels having red, green, and blue values of 255. This result is a very large peak in the histogram at a value of 255.

> *(The bottom histogram in Figure 5 used a brightness value of 0.97 to move the peak at 255 slightly to the left of 255. This caused it to be used to normalize the histogram to a peak value of 100. Having done this, there are hardly any other values of any significance in the color distribution.)*

From this, you might conclude that the methods for enhancing hand-drawn images are probably different from the methods for enhancing images resulting from digital photographs.

Modify a variety of images

If you search the Internet, you should be able to find lots of images that you can download and experiment with. Just remember, if you download a gif image, it will probably contain a lot less color information than a comparable jpg image.

Have fun and learn

Above all, have fun and use this program to learn as much as you can about manipulating images by modifying image pixels using Java.

Test images

Figure 6 contains the jpg file that was used to produce the output shown in Figures 1, 2, and 3. You should be able to right-click on the image in Figure 6 to download and save it locally. Then you should be able to replicate the output produced in Figure 1.

Figure 6 Raw jpg image for Figure 1

A gif image

Figure 7 contains a gif version of the same image as the jpg image in Figure 6. Once again, you should be able to right-click on the image in Figure 7 to download and save it locally. However, if you use it in an attempt to replicate the output shown in Figures 1, 2, and 3, you should notice some major differences resulting from the fact that this is a gif image instead of a jpg image.

Figure 7 Raw gif image for Figure 1

A hand-drawn image

Figure 8 contains a jpg image produced using a typical paint program. You should be able to right-click on the image in Figure 8 to download and save it locally.

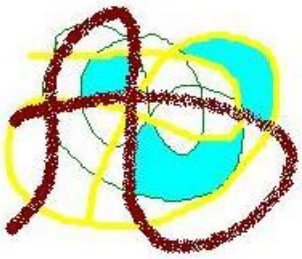

Figure 8 Raw jpg image produced with a paint program

Summary

I showed you how to control the contrast and brightness of an image by controlling the distribution of the color pixels.

A wide distribution exhibits a large amount of contrast. A very narrow distribution results in a washed-out image with very little contrast.

A distribution with a mean value near 255 results in a bright image. A distribution with a mean value near zero results in a dark image.

A gif image contains less color information than a jpg image and is probably less susceptible to enhancement through image manipulation.

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:

- Blur all or part of an image.
- Deal with the effects of noise in 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.
- Morph one image into another image.
- Rotate an image.
- Change the size of an image.
- 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 programs discussed in this lesson are provided in Listing 20 and 21.

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, the Java Advanced Imaging API *(JAI)* has a number of built-in special effects if you prefer to write your own production image-processing programs using Java.

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

```
/*File ImgMod23.java
Copyright 2004, R.G.Baldwin

Illustrates contrast and brightness control.

This program is designed to be driven by the
program named ImgMod02.  Enter the following at
the command line to run this program.

java ImgMod02 ImgMod23 imageFileName

The purpose of this program is to illustrate how
```

to modify the contrast and the brightness of an image by modifying the distribution of color values.

The contrast of an image is determined by the width of the distribution of the color values belonging to the image. If all color values are grouped together in a very tight distribution, the details in the image will tend to be washed out and the overall appearance of the image will tend toward some shade of gray. The shade will depend on the location of the grouping between the extremes of 0 and 255.

At the extremes, if the color values are closely grouped near zero, the colors will range from black to dark gray. If the color values are grouped near 255, the colors will range from very light gray to white.

The contrast of the image can be increased by increasing the width of the distribution of the color values. The contrast can be decreased by decreasing the width of the distribution.

The overall brightness of an image is determined by the location of the grouping of color values. If the grouping tends to be near the upper limit of 255, the image will tend to be very bright. If the grouping tends to be near the lower limit of 0, the image will tend to be very dark.

The brightness of an image can be increased by moving the grouping toward 255, and can be decreased by moving the grouping toward 0.

A straightforward way to change the width of the distribution with, or without changing the general location of the distribution consists of the following steps:
* Calculate the mean or average value of all the color values.
* Subtract the mean from every color value, causing the mean value to be shifted to zero.
* Multiply every color value by the same scale factor. If the scale factor is greater than 1.0, the width of the distribution will be increased. If the scale factor is less than 1.0, the width of the distribution will be decreased.
* Add the original mean value, or a new mean value to every color value to restore the distribution to its original location or to move it to a new location.

The above steps will change the contrast of the image, and may, or may not change its

brightness depending on the mean value that is added in the final step.

A straightforward way to change the location of the distribution is to add (or subtract) the same constant to every color value. This will change the brightness without changing the contrast.

After performing these operations, it is important to make certain that all color values fall within the range of an unsigned byte. This requires eliminating all color values that are greater than 255 and all color values that are less than 0. A simple way to do this is to simply clip the color values at those two limits if they fall outside the limits.

Remember, however, that clipping can change the width, the shape, and the mean value of the distribution if the location of the distribution is near the lower or upper limit. Clipping values to the limits will tend to narrow the distribution and to create a spike in the distribution at the value of the limit.

There are several ways to measure the width of a distribution. One way is to measure the distance between the minimum value and the maximum value. This is not a very good way because a couple of outliers can lead to erroneous conclusions regarding the width of the distribution.

A better way is to compute the root mean square (rms) value of all the color values in the distribution. This approach is less sensitive to outliers and produces results that are more representative of the bulk of the distribution. For distributions of known shapes, it is possible to say what percentage of the color values fall within a range bounded by plus and minus the rms value. For example, if the distribution is uniform (which it probably isn't), approximately 60-percent of all the color values will fall within the range bounded by the rms value on either side of the mean.

If the shape of the distribution is not known, we can only say that for an image containing a large number of different color values, a large percentage of those color values will lie within the range bounded by plus and minus the rms value.

The rms value is computed and displayed in this program solely to provide information to the

```
user.  The rms value is not used in the
computations that control the contrast and
brightness of the image.  For most images, the
user should be able to see a direct
correspondence between the rms value and the
contrast.  For small rms values, the contrast in
the image should appear to be washed out and the
overall color of the image should tend towards
gray.

The program provides a GUI with two text fields
and two histograms.

The text fields make it possible for the user
to modify the contrast and brightness of the
processed image by entering new values for
Contrast and Brightness and pressing the Replot
button on the main display.

The values entered into the text fields are
multiplicative factors.  The initial value in
each text field is 1.0.  For these values, the
processed image should be identical to the
new image.

To increase the contrast or the brightness, type
a value greater than 1.0 into the corresponding
text field and press the Replot button.

To decrease the contrast or the brightness, type
a value less than 1.0 into the text field and
press the Replot button.

It isn't necessary to press the Enter key to type
the value into the text field, but doing so won't
cause any harm.

Entering a value that cannot be converted to a
value of type double will cause the program to
throw an exception.

The top histogram shows the distribution of
color values for the original image.  The bottom
histogram shows the distribution of color values
for the modified image.

The histogram values are normalized to a peak
value of 100, exclusive of the values at 0 and
255.  These two values can, and often will
exceed 100 and go out of plotting range on the
histograms.

Tested using SDK 1.4.2 and WinXP
*******************************************/
import java.awt.*;
import javax.swing.Box;
```

```
import javax.swing.BoxLayout;

class ImgMod23 extends Frame
                           implements ImgIntfc02{

  TextField contrastField;
  TextField brightField;
  Panel input;
  OrigHistogramPanel origHistPanel;
  NewHistogramPanel newHistPanel;
  int[] origHistogram = new int[256];
  int[] newHistogram = new int[256];


  //Constructor must take no parameters
  ImgMod23(){
    //Create a Box container with a vertical
    // layout and place it in the center of
    // the Frame.
    Box aBox = new Box(BoxLayout.Y_AXIS);
    this.add(aBox,BorderLayout.CENTER);

    //Create and place the user input panel at
    // the top of the vertical stack.  Make it
    // yellow
    input = new Panel();

    //Create panels to group the labels with the
    // text fields and add them to the input
    // panel under FlowLayout.
    Panel contrastPanel = new Panel();
    contrastPanel.add(new Label("Contrast"));
    contrastField = new TextField("1.0",5);
    contrastPanel.add(contrastField);
    input.add(contrastPanel);

    Panel brightnessPanel = new Panel();
    brightnessPanel.add(new Label("Brightness"));
    brightField = new TextField("1.0",5);
    brightnessPanel.add(brightField);
    input.add(brightnessPanel);

    input.add(new Label(
                      "Press Replot to change"));

    input.setBackground(Color.YELLOW);
    aBox.add(input);

    //Create and place the panel for the
    // original histogram in the middle of the
    // stack.  Make it green.
    origHistPanel = new OrigHistogramPanel();
    origHistPanel.setBackground(Color.GREEN);
    aBox.add(origHistPanel);

    //Create and place the panel for the new
```

```
    // histogram at bottom of the stack.  Make
    // it orange.
    newHistPanel = new NewHistogramPanel();
    newHistPanel.setBackground(Color.ORANGE);
    aBox.add(newHistPanel);

    //Set miscellaneous properties.
    setTitle("Copyright 2004, Baldwin");
    setBounds(400,0,275,400);
    setVisible(true);
  }//end constructor
  //--------------------------------------//

  //An inner class for the original histogram
  // panel.  This is necessary to make it
  // possible to override the paint method.
  class OrigHistogramPanel extends Panel{
    public void paint(Graphics g){
      //Following constant corrects for positive
      // direction on the y-axis.
      final int flip = 110;
      //Following constant is used to shift the
      // histogram plot 5 pixels to the right.
      final int shift = 5;
      //Draw the horizontal axis
      g.drawLine(0 + shift,flip,
                                255 + shift,flip);
      //Draw the histogram
      for(int cnt = 0;cnt < origHistogram.length;
                                              cnt++){
        g.drawLine(cnt + shift,flip - 0,
                    cnt + shift,
                      flip - origHistogram[cnt]);
      }//end for loop
    }//end paint
  }//end class OrigHistogramPanel
  //--------------------------------------//

  //An inner class for the new histogram
  // panel.  This is necessary to make it
  // possible to override the paint method.
  class NewHistogramPanel extends Panel{
    public void paint(Graphics g){
      //Following constant corrects for positive
      // direction on the y-axis.
      final int flip = 110;
      //Following constant is used to shift the
      // histogram plot 5 pixels to the right.
      final int shift = 5;
      //Draw the horizontal axis
      g.drawLine(0 + shift,flip,
                                255 + shift,flip);
      //Draw the histogram
      for(int cnt = 0;cnt < newHistogram.length;
                                              cnt++){
        g.drawLine(cnt + shift,flip - 0,
```

```
                    cnt + shift,
                      flip - newHistogram[cnt]);
      }//end for loop
    }//end paint
  }//end class NewHistogramPanel
  //-----------------------------------------//

  //This method is required by ImgIntfc02.
  public int[][][] processImg(
                            int[][][] threeDPix,
                            int imgRows,
                            int imgCols){

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

    //Get user input values for contrast and
    // brightness.  These values will be used as
    // multipliers to change the contrast and
    // the brightness.
    double contrast = Double.parseDouble(
                        contrastField.getText());
    double brightness = Double.parseDouble(
                          brightField.getText());

    //Make a working copy of the 3D array to
    // avoid making permanent changes to the
    // original image data.
    int[][][] output3D =
                      new int[imgRows][imgCols][4];
    for(int row = 0;row < imgRows;row++){
      for(int col = 0;col < imgCols;col++){
        output3D[row][col][0] =
                            threeDPix[row][col][0];
        output3D[row][col][1] =
                            threeDPix[row][col][1];
        output3D[row][col][2] =
                            threeDPix[row][col][2];
        output3D[row][col][3] =
                            threeDPix[row][col][3];
      }//end inner loop
    }//end outer loop


    //Get, save, display, and remove the mean.
    int mean = getMean(output3D,imgRows,imgCols);
    System.out.println("Original mean: " + mean);
    removeMean(output3D,imgRows,imgCols,mean);

    //Get and display the rms value.  The rms
    // value is for user information only.  It
    // is not actually used by the program.
    int rms = getRms(output3D,imgRows,imgCols);
    System.out.println("Original rms: " + rms);

    //Scale each color value by the contrast
```

```
    // multiplier.  This will either expand or
    // compress the distribution.
    scale(output3D,imgRows,imgCols,contrast);
    System.out.println("New rms"
             + getRms(output3D,imgRows,imgCols));

    //Restore the mean to a non-zero value by
    // adding the same value to each color value.
    // The value added is the product of the
    // new mean and the brightness multiplier.
    shiftMean(output3D,imgRows,imgCols,
                          (int)(brightness*mean));
    System.out.println("New mean : "
            + getMean(output3D,imgRows,imgCols));

    //Clip all color values at 0 and 255 to make
    // certain that no color value is out of the
    // range of an unsigned byte.
    clip(output3D,imgRows,imgCols);
    System.out.println();

    //Create and draw the two histograms
    origHistogram = getHistogram(threeDPix,
                                 imgRows,imgCols);
    origHistPanel.repaint();

    newHistogram = getHistogram(output3D,
                                 imgRows,imgCols);
    newHistPanel.repaint();

    //Return the modified 3D array of pixel data.
    return output3D;

  }//end processImg
  //-----------------------------------------//

  //Method to create a histogram and return it in
  // an array of type int.  The histogram is
  // normalized to a peak value of 100 exclusive
  // of the values at 0 and 255.  Those values
  // can, and often do exceed 100.
  int[] getHistogram(int[][][] data3D,
                     int imgRows,int imgCols){
    int[] hist = new int[256];
    for(int row = 0;row < imgRows;row++){
      for(int col = 0;col < imgCols;col++){
        hist[data3D[row][col][1]]++;
        hist[data3D[row][col][2]]++;
        hist[data3D[row][col][3]]++;
      }//end inner for loop
    }//end outer for loop
    //Get the maximum value, exclusive of the
    // values at 0 and 255
    int max = 0;
    for(int cnt = 1;cnt < hist.length - 1;cnt++){
      if(hist[cnt] > max){
```

```
        max = hist[cnt];
      }//end if
    }//end for loop

    //Normalize histogram to a peak value of 100
    // based on the max value, exclusive of the
    // values at 0 and 255
    for(int cnt = 0;cnt < hist.length;cnt++){
      hist[cnt] = 100 * hist[cnt]/max;
    }//end for loop
    return hist;
  }//end getHistogram
  //-----------------------------------------//

  //Method to calculate and return the mean
  // of all the color values
  int getMean(int[][][] data3D,int imgRows,
                                    int imgCols){

    int pixelCntr = 0;
    long accum = 0;
    for(int row = 0;row < imgRows;row++){
      for(int col = 0;col < imgCols;col++){
        accum += data3D[row][col][1];
        accum += data3D[row][col][2];
        accum += data3D[row][col][3];
        pixelCntr += 3;
      }//end inner for loop
    }//end outer for loop

    return (int)(accum/pixelCntr);

  }//end getMean
  //-----------------------------------------//

  //Method to remove the mean causing the new
  // mean value to be 0
  void removeMean(int[][][] data3D,int imgRows,
                            int imgCols,int mean){
    for(int row = 0;row < imgRows;row++){
      for(int col = 0;col < imgCols;col++){
        data3D[row][col][1] -= mean;
        data3D[row][col][2] -= mean;
        data3D[row][col][3] -= mean;
      }//end inner for loop
    }//end outer for loop
  }//end removeMean
  //-----------------------------------------//

  //Method to calculate the root mean square
  // value of all the color values.
  int getRms(int[][][] data3D,int imgRows,
                                    int imgCols){
    int pixelCntr = 0;
    long accum = 0;
    for(int row = 0;row < imgRows;row++){
```

```
      for(int col = 0;col < imgCols;col++){
        accum += data3D[row][col][1] *
                                data3D[row][col][1];
        accum += data3D[row][col][2] *
                                data3D[row][col][2];
        accum += data3D[row][col][3] *
                                data3D[row][col][3];
        pixelCntr += 3;
      }//end inner for loop
    }//end outer for loop
    int meanSquare = (int)(accum/pixelCntr);
    int rms = (int)(Math.sqrt(meanSquare));
    return rms;
  }//end getRms
  //----------------------------------------//

  //Method to scale the data and expand or
  // compress the distribution
  void scale(int[][][] data3D,int imgRows,
                       int imgCols,double scale){
    for(int row = 0;row < imgRows;row++){
      for(int col = 0;col < imgCols;col++){
        data3D[row][col][1] *= scale;
        data3D[row][col][2] *= scale;
        data3D[row][col][3] *= scale;
      }//end inner for loop
    }//end outer for loop
  }//end scale
  //----------------------------------------//

  //Method to shift the mean to a new value.
  void shiftMean(int[][][] data3D,int imgRows,
                         int imgCols,int newMean){
    for(int row = 0;row < imgRows;row++){
      for(int col = 0;col < imgCols;col++){
        data3D[row][col][1] += newMean;
        data3D[row][col][2] += newMean;
        data3D[row][col][3] += newMean;
      }//end inner for loop
    }//end outer for loop
  }//end shiftMean
  //----------------------------------------//

  //Method to clip the color data at 0 and 255
  void clip(int[][][] data3D,int imgRows,
                                     int imgCols){
    for(int row = 0;row < imgRows;row++){
      for(int col = 0;col < imgCols;col++){
        if(data3D[row][col][1] < 0)
          data3D[row][col][1] = 0;
        if(data3D[row][col][1] > 255)
          data3D[row][col][1] = 255;

        if(data3D[row][col][2] < 0)
          data3D[row][col][2] = 0;
        if(data3D[row][col][2] > 255)
```

```
          data3D[row][col][2] = 255;

        if(data3D[row][col][3] < 0)
          data3D[row][col][3] = 0;
        if(data3D[row][col][3] > 255)
          data3D[row][col][3] = 255;

      }//end inner for loop
    }//end outer for loop
  }//end clip
  //----------------------------------------//

}//end class ImgMod23
```

Listing 20

```
/*File ImgMod02a.java
Copyright 2004, R.G.Baldwin

This is an update of the program named ImgMod02.
This update is designed to use
BufferedImage.getRaster in place of PixelGrabber.

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 ImgMod02a ProcessProgramName 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 curent 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.*;
import com.sun.image.codec.jpeg.*;
import java.io.*;

class ImgMod02a extends Frame{
  Image rawImg;
  BufferedImage buffImage;
  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
      // informtion 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
    ImgMod02a obj = new ImgMod02a();
  }//end main
  //---------------------------------------//

  public ImgMod02a(){//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];

    //Create an empty BufferedImage object
    buffImage = new BufferedImage(
                    imgCols,
                    imgRows,
                    BufferedImage.TYPE_INT_ARGB);

    // Draw Image into BufferedImage
```

```
    Graphics g = buffImage.getGraphics();
    g.drawImage(rawImg, 0, 0, null);

    //Convert the BufferedImage to numeric pixel
    // representation.
    DataBufferInt dataBufferInt =
            (DataBufferInt)buffImage.getRaster().
                                    getDataBuffer();
    oneDPix = dataBufferInt.getData();

    //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 a counterfit 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 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

    //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 ImgMod02a.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
// ImgMod02a.

//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 21

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-