Processing Image Pixels, Color Intensity, Color Filtering, and Color Inversion

Learn to write a Java program to control color intensity, apply color filtering, and apply color inversion to an image. Learn to relate the colors to points on a color wheel and points in a color cube.

Published: June 14, 2005
By Richard G. Baldwin

Java Programming, Notes # 406

Preface

Fourth in a series

This is the fourth 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: Controlling Contrast and Brightness. 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 processed image along with the original image.

The lesson entitled Processing Image Pixels Using Java: Controlling Contrast and Brightness provided an upgraded version of that program named **ImgMod02a**. **ImgMod02a** serves as a driver that controls the execution of a second program that actually processes the pixels.

The program that I will explain in this lesson runs under the control of **ImgMod02a**. In order to compile and run the program that I will provide in this lesson, you will need to go to the lessons entitled Processing Image Pixels Using Java: Controlling Contrast and Brightness and Processing Image Pixels using Java, Getting Started to get copies of the program named **ImgMod02a** and the interface named **ImgIntfc02**.

Purpose of this lesson

The purpose of this lesson is to teach you how to write a Java program that can be used to:

- Control color intensity
- Apply color filtering
- Apply color inversion

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.

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.

Sample program output

I will begin this lesson by showing you three examples of the types of things that you can do with this program. I will discuss the examples very briefly here and will discuss them in more detail later in the lesson.

Color intensity control

Figure 1 shows an example of color intensity control. The bottom image in Figure 1 is the result of reducing the intensity of every color pixel to fifty-percent of its original value. As you can see, this basically caused the intensity of the entire image to be reduced resulting in a darker image where the colors were somewhat washed out.

Figure 1 Color intensity control example.

The user interface GUI

Figure 2 shows the state of the user interface GUI that produced Figure 1. Each of the three sliders in Figure 2 controls the intensity of one of the colors red, green, and blue. The intensity of each color can be adjusted within the range from 0% to 100% of its original value.

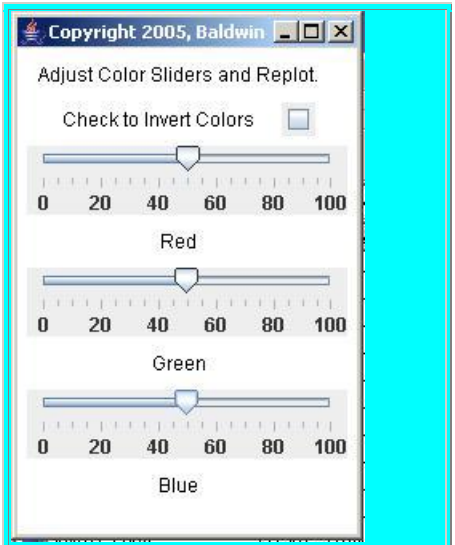

Figure 2 Control for Fig. 1

Each of the sliders in Figure 2 was adjusted to a value of 50, causing the intensity of every color in every pixel to be reduced to 50% of its original value.

> *(Note that the check box at the top was not checked. I will explain the purpose of this checkbox later.)*

Color filtering

Figure 3 shows an extreme example of color filtering.

> *(I elected to provide an extreme example so that the results would be obvious.)*

In Figure 1, there was no modification of any color relative to any other color. *(The value of every color was adjusted to 50% of its original value.)* However, in Figure 3, the relative intensities of the three colors were modified relative to each other.

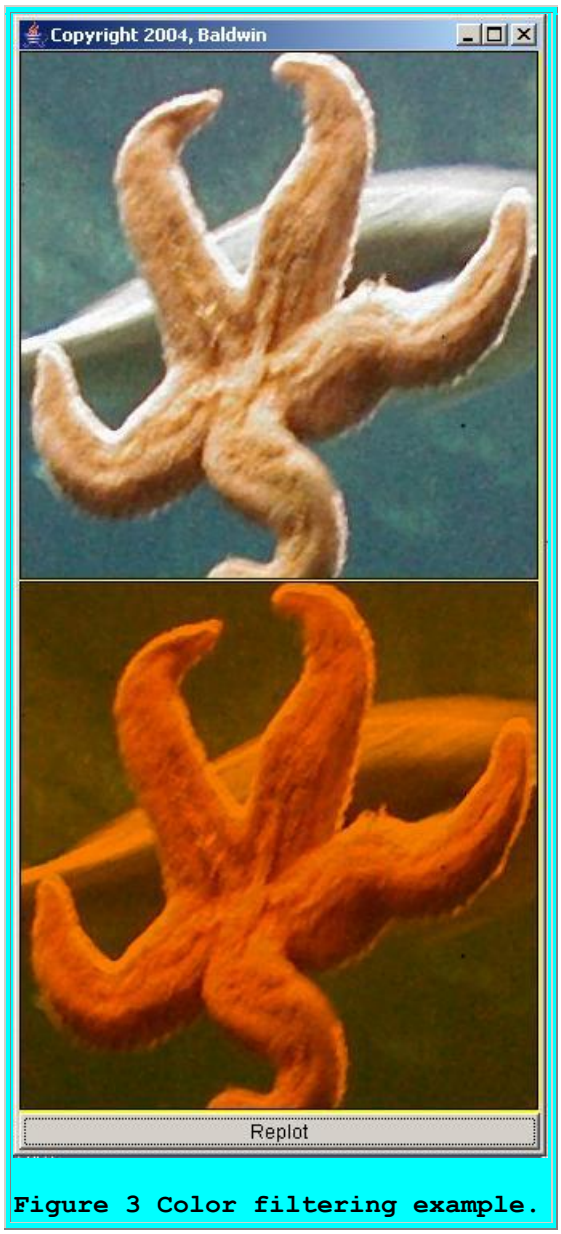

Figure 3 Color filtering example.

There was no change to the color values for any of the red pixels in Figure 3. The color values for all of the green pixels were reduced to 50% of their original values. The color values for all blue pixels were reduced to zero. Thus, the color blue was completely eliminated from the output.

As you can see, modifying the pixel color values in this way caused the overall color of the processed image to be more orange than the original.

> *(Some would say that the processed image in Figure 3 is warmer than the original image in Figure 3 because it emphasizes warm colors rather than cool colors.)*

The user interface GUI for Figure 3

Figure 4 shows the state of the user interface GUI that produced Figure 3.

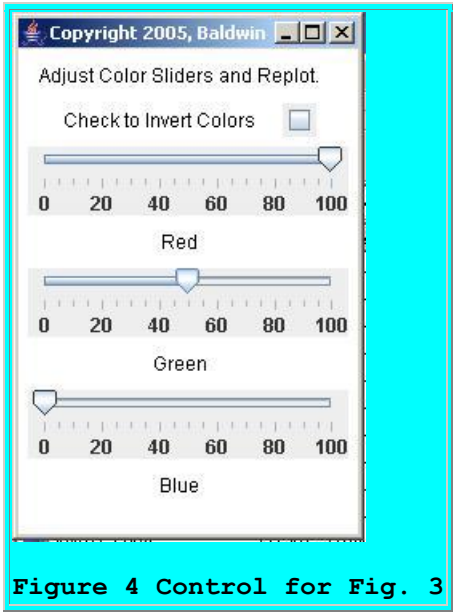

Figure 4 Control for Fig. 3

The red slider in Figure 4 is positioned at 100, causing the red color values of all the pixels to remain unchanged. The green slider is positioned at 50, causing the green color values of all the pixels to be reduced to 50% of their original values. The blue slider is positioned at 0 causing the blue color values of all pixels to be reduced to 0.

Once again the checkbox at the top of Figure 4 is not checked. I will explain the purpose of this checkbox in the next section.

Color inversion

Figure 5 shows an example of color inversion with no color filtering.

> *(Note that it is also possible to apply a combination of color filtering and color inversion.)*

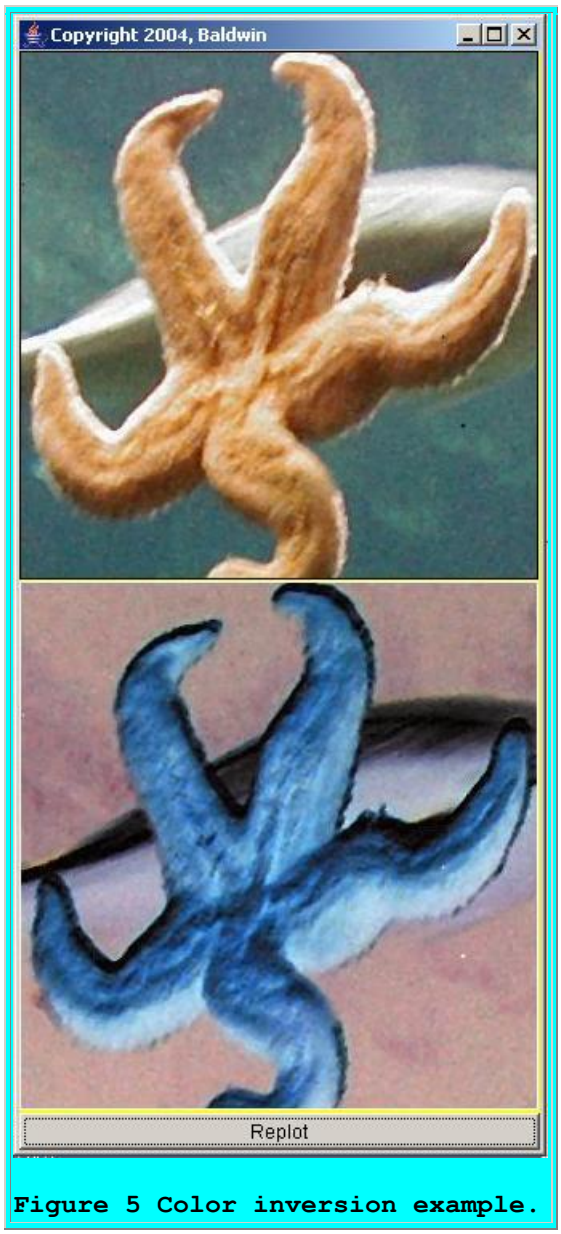

Figure 5 Color inversion example.

What is color inversion?

I will have a great deal to say about color inversion later in this lesson. For now, suffice it to say that color inversion causes a change to all the colors in an image. That change is computationally economical, reversible, and usually obvious to the viewer. As you can readily see, the colors in the processed image in Figure 5 are obviously different from the colors in the original image.

The user interface GUI for Figure 5

Figure 6 shows the state of the user interface GUI that produced Figure 5.

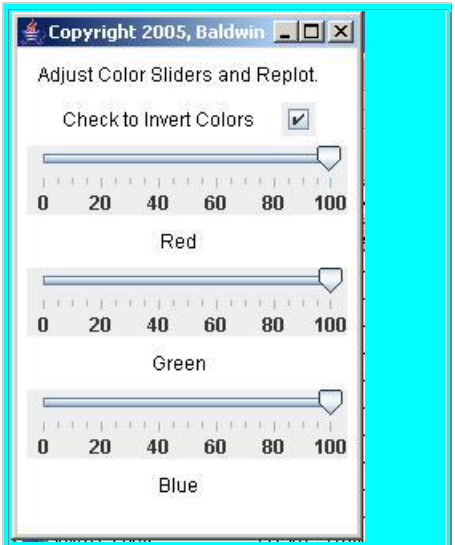

Figure 6 Control for Fig. 5

The check box at the top of Figure 6 is checked, sending a message to the image-processing program to implement color inversion.

Each of the sliders in Figure 6 is positioned at 100. As a result, no color filtering was applied. As mentioned earlier, however, it is possible to combine color filtering with color inversion. In fact, by using comment indicators to enable and disable different blocks of code and recompiling, the program that I will discuss later makes it possible to combine color filtering and color inversion in two different ways:

- Filter first and then invert.
- Invert first and then filter.

The two different approaches can result in significantly different results.

Display format

The images shown in Figures 1, 3, and 5 were produced by the driver program named **ImgMod02a**. The user interface GUIs in Figures 2, 4, and 6 were produced by the program named **ImgMod15**.

As in all of the graphic output produced by the driver program named **ImgMod02a**, the original image is shown at the top and the processed image is shown at the bottom.

An interactive image-processing program

The image-processing program named **ImgMod15** illustrated by the above figures allows the user to interactively

- Control the color intensity

- Apply color filtering
- Apply color inversion

Color intensity and color filtering are controlled by adjusting the three sliders where each slider corresponds to one of the colors red, green, and blue.

Color inversion is controlled by checking or not checking the check box near the top of the GUI.

After making adjustments to the GUI, the user presses the **Replot** button shown at the bottom of Figures 1, 3, and 5 to cause the image to be reprocessed and replotted.

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.

File formats

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 **ImgMod02a** is compatible with **gif** files, **jpg** files, and possibly some other file formats as well.

A three-dimensional array of pixel data as type int

The driver program named **ImgMod02a**:

- 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.

Display of processed image results

When the image-processing program completes its work, the driver program named **ImgMod02a**:

- Receives a reference to a three-dimensional array object containing processed pixel data from the image-processing program.

- Displays the original image and the processed image in a stacked display as shown in Figure 1.

Reprocessing with different parameters

In addition, the way in which the two programs work together makes it possible for the user to:

- Provide new input data to the image-processing program.
- Invoke the image-processing program again.
- Create a new display showing the newly-processed image along with the original image.

The manner in which all of this communication between the programs 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 lesson will show you how to write an image-processing program that receives raw pixel data in the form of a three-dimensional array of type **int**, and returns processed pixel data in the form of a three-dimensional array of type **int**. The program is designed to achieve the image-processing objectives described earlier.

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 amounts 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.

Preview

Three programs and one interface

The program that I will discuss in this lesson requires the program named **ImgMod02a** and the interface named **ImgIntfc02** for compilation and execution. I provided and explained that material in the earlier lessons entitled Processing Image Pixels using Java, Getting Started and Processing Image Pixels Using Java: Controlling Contrast and Brightness.

I will present and explain a new Java program named **ImgMod15** in this lesson. This program, when run under control of the program named **ImgMod02a**, will produce outputs similar to those shown in Figures 1, 3, and 5.

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

I will also provide, *(but will not explain)* a simple program named **ImgMod27**. This program can be used to display *(in 128 different panels)* all of the 16,777,216 different colors that can be produced using three primary colors, each of which can take on any one of 256 values. The different colors are displayed in groups of 131,072 colors in each panel.

The processImg method

The program named **ImgMod15**, *(and all image-processing programs that are capable of being driven by **ImgMod02a**)*, 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 **ImgMod02a**, 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 must return a three-dimensional array containing the pixel data for a processed version of the original image.

A *before* and *after* display

When the **processImg** method returns, the driver program named **ImgMod02a** causes the original image and the processed image to be displayed in a frame with the original image above the processed image *(see Figure 1 for an example of the display format).*

Usage information for ImgMod02a and ImgMod15

To use the program named **ImgMod02a** to drive the program named **ImgMod15**, enter the following at the command line:

```
java ImgMod02a ImgMod15 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 images in Figures 16, 17, and 18 to download and save the images used in this lesson. Then you should be able to replicate the results shown in the various figures in this lesson.)*

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 **ImgMod15** provides a GUI for user input, as shown in Figure 2. The sliders on the GUI make it possible for the user to provide different filter values for red, green, and blue each time the image-processing method is rerun. The check box near the top of the GUI makes it possible for the user to request that the colors in the image be inverted.

To rerun the image-processing method with different parameters, adjust the sliders, optionally check the check box in the GUI, and then press the **Replot** button at the bottom of the main display.

Discussion and Sample Code

The program named ImgMod15

This program illustrates how to control color intensity, apply color filters, and apply color inversion to an image.

The program is designed to be driven by the program named **ImgMod02a**.

The *before* and *after* images

The program places two GUIs on the screen. One GUI displays the "before" and "after" versions of an image that is subjected to color intensity control, color filtering, and color inversion.

The image at the top of this GUI is the "before" image. The image at the bottom is the "after" image. An example is shown in Figure 1.

The user interface GUI

The other GUI provides instructions and components by which the user can control the processing of the image. An example of the user interface GUI is shown in Figure 2.

A check box appears near the top of this GUI. If the user checks the check box, color inversion is performed. If the check box is not checked, no color inversion is performed.

This GUI also provides three sliders that make it possible for the user to control color intensity and color filtering. Each slider controls the intensity of a single color. The intensity control ranges from 0% to 100% of the original intensity value for each color for every pixel.

Controlling color intensity

If all three sliders are adjusted to the same value and the replot button is pressed, the overall intensity of the image is modified with no change in the relative contribution of each color. This makes it possible to control the overall intensity of the image from very dark *(black)* to the maximum intensity supported by the original image. This is illustrated in Figure 1.

Color filtering

If the three sliders are adjusted to different values and the replot button is pressed, color filtering occurs. In this case, the intensity of each color is changed relative to the intensity of the other colors. This makes it possible, for example to adjust the "warmth" of the image by emphasizing red over blue, or to make the image "cooler" by emphasizing blue over red. This is illustrated in Figure 3.

A greenscale image

It is also possible to totally isolate and view the individual contributions of red, green, and blue to the overall image as illustrated in Figure 7.

The values for red and blue were set to zero for all of the pixels in the processed image in Figure 7. This leaves only the differing green values for the individual pixels, producing what might be thought of as a *greenscale* image *(in deference to the use of the term grayscale for a common class of black, gray, and white images).*

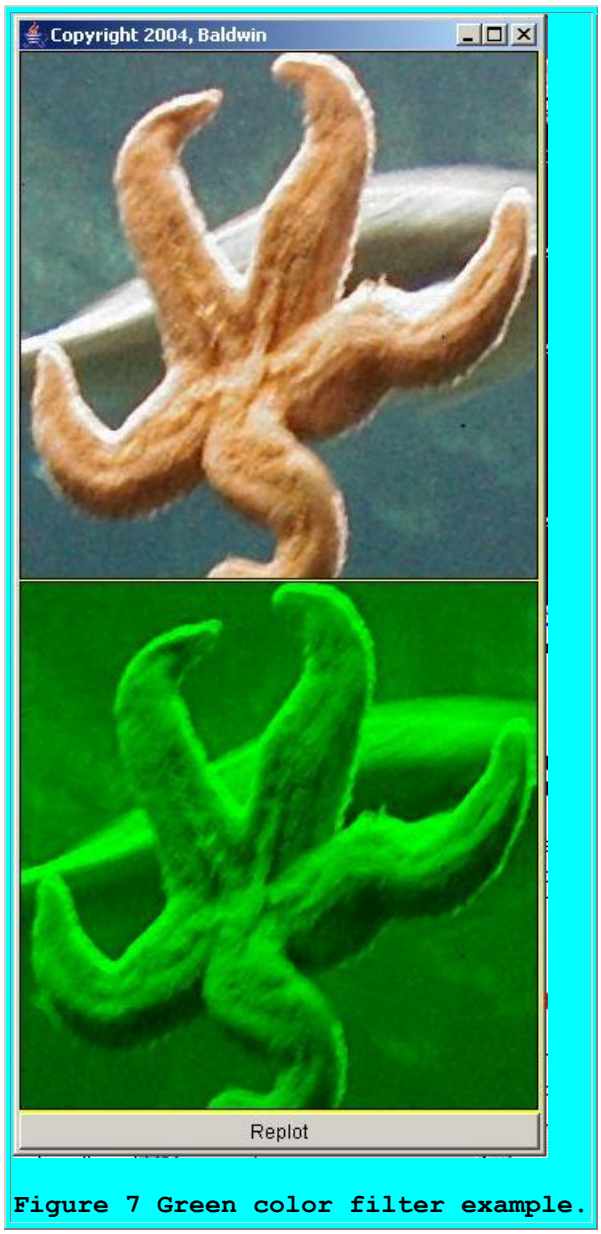

Figure 7 Green color filter example.

The user interface GUI for Figure 7

Figure 8 shows the state of the user interface GUI that produced the processed image in Figure 7. As you can see, the sliders for red and blue were set to zero causing all red and blue color values to be set to zero. The slider for green was set to 100 causing the green value for every pixel to remain the same as in the original image.

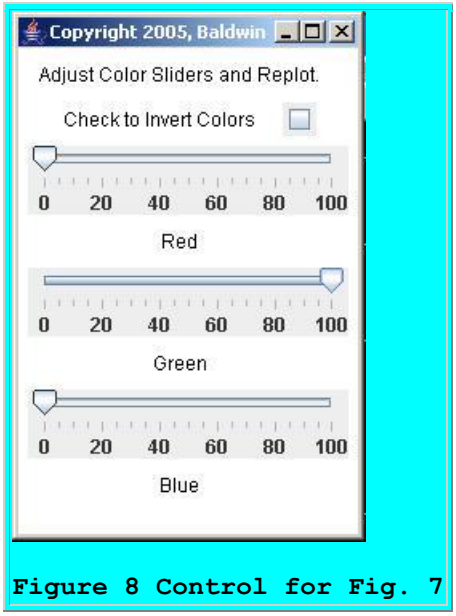

Figure 8 Control for Fig. 7

The checkbox was not checked. Therefore, color inversion was not performed.

Which comes first, the filter or the inversion?

As written, the program applies color filtering before it applies color inversion. As you will see later, sample code is also provided that can be used to modify the program to cause it to provide color inversion before it applies color filtering. There is a significant difference in the results produced by these two approaches, and you may want to experiment with them.

A practical example of color inversion

As a side note, Microsoft Word and Microsoft FrontPage appear to use color inversion to change the colors in images that have been selected for editing. I will have more to say about this later.

Beware of transparent images

This program illustrates the modification of red, green, and blue values belonging to all the pixels in an image. It works best with an image that contains no transparent areas. The pixel modifications performed in this program have no impact on transparent pixels. Therefore, if you don't see what you expect when you process an image, it may be because your image contains transparent pixels.

Will discuss in fragments

I will break the program down into fragments for discussion. A complete listing of the program is provided in Listing 8 near the end of the lesson.

The ImgMod15 class

The ImgMod15 class begins in Listing 1. In order to be suitable for being driven by the program named **ImgMod02a**, this class must implement the interface named **ImgIntfc02**.

```
class ImgMod15 extends Frame
                            implements
ImgIntfc02{

  //GUI components used to control
color
  // filtering and color inversion.
  JSlider redSlider;
  JSlider greenSlider;
  JSlider blueSlider;
  JCheckBox checkBox;
```

Listing 1

The class extends **Frame**, because an object of this class is the user interface GUI shown in Listings 2, 4, 6, and 8. The code in Listing 1 declares four instance variables that will refer to the check box and the three sliders in Figure 8.

The constructor for ImgMod15

The constructor is shown in its entirety in Listing 2. Because of the way that an object of the class is instantiated by **ImgMod02a**, the constructor is not allowed to take any parameters.

```
  ImgMod15(){
    //Create and display the user-
input GUI.
    setLayout(new FlowLayout());

    //Provide user instructions at the
top of the
    // GUI.
    add(new Label(
            "Adjust Color Sliders and
Replot."));

    //Provide a check box that is used
to request
    // color inversion.
    checkBox = new JCheckBox();
    add(new Label("Check to Invert
Colors"));
    add(checkBox);

    //Create three sliders each with a
range of
    // 0 to 100 and an initial value
of 100.
```

```
    redSlider = new
JSlider(0,100,100);
    add(redSlider);
    //Put a label under the slider.
    add(new Label("Red"));

    greenSlider = new
JSlider(0,100,100);
    add(greenSlider);
    add(new Label("Green"));

    blueSlider = new
JSlider(0,100,100);
    add(blueSlider);
    add(new Label("Blue"));

    //Put numeric labels and tick
marks on the
    // sliders.
    redSlider.setMajorTickSpacing(20);
    redSlider.setMinorTickSpacing(5);
    redSlider.setPaintTicks(true);
    redSlider.setPaintLabels(true);


greenSlider.setMajorTickSpacing(20);

greenSlider.setMinorTickSpacing(5);
    greenSlider.setPaintTicks(true);
    greenSlider.setPaintLabels(true);


blueSlider.setMajorTickSpacing(20);
    blueSlider.setMinorTickSpacing(5);
    blueSlider.setPaintTicks(true);
    blueSlider.setPaintLabels(true);

    setTitle("Copyright 2005,
Baldwin");
    setBounds(400,0,220,330);
    setVisible(true);
  }//end constructor

Listing 2
```

Although the code in Listing 2 is rather long, all of the code in Listing 2 is straightforward if you are familiar with the construction of GUIs in Java. If you are not familiar with such constructions, you should study some of my other lessons on this topic. As mentioned earlier, you will find an index to all of my lessons at www.DickBaldwin.com.

The processImg method

To be compatible with **ImgMod02a**, 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.

The beginning of the **processImg** method is shown in Listing 3.

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

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

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

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

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

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

Listing 3

It's best to make and modify a copy

Normally the **processImg** method should make a copy of the incoming array and process the copy rather than modifying the original. Then the method should return a reference to the processed copy of the three-dimensional pixel array. The code in Listing 3 makes such a copy.

Get the slider values

The code in Listing 4 gets the current values of each of the three sliders. This information will be used to scale the red, green, and blue pixel values to new values in order to implement color intensity control and color filtering. The new color values can range from 0% to100% of the original values

```
    int redScale =
redSlider.getValue();
    int greenScale =
greenSlider.getValue();
    int blueScale =
blueSlider.getValue();
```

Listing 4

Process each color value

The code in Listing 5 is the beginning of a **for** loop that is used to process each color value for every pixel. The boldface code in Listing 5 is executed for the case where the check box near the top of Figure 2 has not been checked.

```
    for(int row = 0;row <
imgRows;row++){
      for(int col = 0;col <
imgCols;col++){
        if(!checkBox.isSelected()){
          //Apply color filtering but
no color
          // inversion
          temp3D[row][col][1] =
              temp3D[row][col][1] *
redScale/100;
          temp3D[row][col][2] =
            temp3D[row][col][2] *
greenScale/100;
          temp3D[row][col][3] =
             temp3D[row][col][3] *
blueScale/100;
```

Listing 5

In this case, each color value for every pixel is multiplied by a scale factor that is determined by the position of the slider corresponding to that color. In effect, the product of the color value and

the scale factor causes the processed color value to range from 0% to 100% of the original color value.

Note that the code in Listing 5 is the first half of an **if-else** statement.

Apply color inversion

In the event that the color-inversion check box is checked, the boldface code in Listing 6 is executed instead of the boldface code in Listing 5. The code in Listing 6 first applies color filtering using the slider values and then applies color inversion.

```
        }else{
          temp3D[row][col][1] = 255 -
              temp3D[row][col][1] *
redScale/100;
          temp3D[row][col][2] = 255 -
            temp3D[row][col][2] *
greenScale/100;
          temp3D[row][col][3] = 255 -
             temp3D[row][col][3] *
blueScale/100;

Listing 6
```

The formula for color inversion

Recall that an individual color value can fall anywhere in the range from 0 to 255. The code in Listing 6 performs color inversion by subtracting the scaled color value from 255. Therefore, a scaled color value of 200 would be inverted into a value of 55. Likewise, a scaled color value of 55 would be inverted into a value of 200. Thus, the inversion process can be reversed simply by applying it twice in succession.

Since it may not be obvious what the results of such an operation will be, I will discuss the ramifications of color inversion in some detail.

An experiment

Let's begin with an experiment. You will need access to either Microsoft Word or Microsoft FrontPage to perform this experiment.

Get and save the image

Figure 5 shows the result of performing color inversion on an image of a starfish. The original image is shown at the top of Figure 5 and the color-inverted image is shown at the bottom of Figure 5. Begin the experiment by right-clicking the mouse on the image in Figure 5 and saving the image locally on your disk.

Insert the image into a Word or FrontPage document

Now create a new document in either Microsoft Word or Microsoft FrontPage and type a couple of paragraphs of text into the new document.

Insert the image that you saved between the paragraphs in your document. It should be the image with the tan starfish at the top and the blue starfish at the bottom.

Select the image

Now use your mouse and select some of the text from both paragraphs. Include the image between the paragraphs in the selection. If your system behaves like mine, the starfish at the top should turn blue and the starfish at the bottom should turn tan. In other words, the two images should be exactly the same except that their positions should be reversed.

What does this mean?

Whenever an image is selected in an editor program like Microsoft Word or Microsoft FrontPage, some visual change must be made to the image so that the user will know that the image has been selected. It appears that Microsoft inverts the colors in selected images in Word and FrontPage for this purpose.

> *(Note, however, that the Netscape browser, the Netscape Composer, and the Internet Explorer browser all use a different method for indicating that an image has been selected, so this is not a universal approach.)*

Why use inverted colors?

Color inversion is a very good way to change the colors in a selected image. The approach has several very good qualities.

Computationally economical

To begin with, inverting the colors is computationally economical. All that is required computationally to invert the colors is to subtract each color value from 255. This is much less demanding of computer resources than would be the case if the computation required multiplication or division, for example.

Overflow is not possible

Whenever you modify the color values in a pixel, you must be very careful to make sure that the new color value is within the range from 0 to 255. Otherwise, serious overflow problems can result. The inversion process guarantees that the new color value will fall within this range, so overflow is not possible.

A reversible process

The process is guaranteed to be reversible with no requirement to maintain any information outside the image regarding the original color values in the image. All that is required to restore the inverted color value back to the original color value is to subtract the inverted color value from 255. The original color value is restored after two successive inversions. Thus, it is easy and economical to switch back and forth between original color values and inverted color values.

Given all of the above, I'm surprised that the color-inversion process isn't used by programs other than Word and FrontPage.

Another example of color inversion

The color values in a digitized color film negative are similar to *(but not identical to)* the inverse of the colors in the corresponding color film positive. Therefore, some photo processing programs begin the process of converting a digitized color film negative to a positive by inverting the colors. Additional color adjustments must usually be made after inversion to get the colors just right.

You will find an interesting discussion of this process in an article entitled Converting negative film to digital pictures by Phil Williams.

What will the inverted color be?

Another interesting aspect of color inversion has to do with knowing what color will be produced by applying color inversion to a pixel with a given color. For this, let's look at another example shown in Figures 9 and 10.

Figure 9 shows the result of applying color inversion to the pure primary colors red, green, and blue.

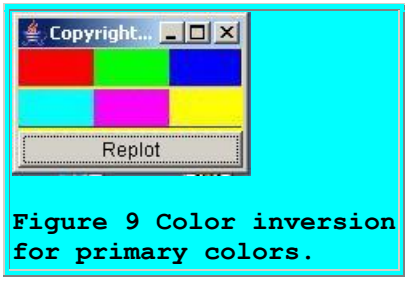

Figure 9 Color inversion for primary colors.

The color bar at the top in Figure 9 shows the three primary colors. The color bar at the bottom shows the corresponding inverted colors.

No color filtering was applied

Figure 10 shows that no color filtering was involved. The colors shown in the bottom image of Figure 9 are solely the result of performing color inversion on the top image in Figure 9.

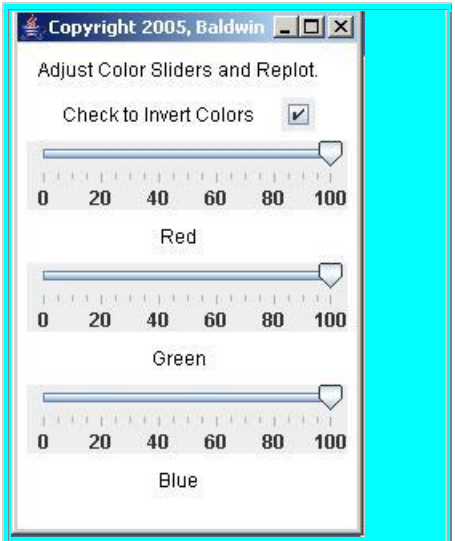

Figure 10 Control for Fig 9

Experimental results

From Figure 9, we can conclude experimentally that applying color inversion to a pure red pixel will cause the new pixel color to be aqua. Similarly, applying color inversion to a pure green pixel will cause the new pixel color to be fuchsia. Finally, applying color conversion to a pure blue pixel will cause the new pixel color to be yellow. To summarize:

- Red inverts to aqua
- Green inverts to fuchsia
- Blue inverts to yellow

An explanation of the results

Consider why the experimental results turn out the way that they do. Consider the case of the pure blue pixel. The red, green, and blue color values for that pixel are as shown below:

- R = 0
- G = 0
- B = 255

Let the inverted color values be given by R', G', and B'. Looking back at the code in Listing 6 *(with no color filtering applied)*, the color values for the pixel following the inversion will be:

- R' = 255 - 0 = 255
- G' = 255 - 0 = 255
- B' = 255 - 255 = 0

The inverted color is yellow

Thus we end up with a pixel having full color intensity for red and green and no intensity for blue. What do we get when we mix red and green in equal amounts? The answer is yellow. Adding equal amounts of red and green produces yellow. Hence, the inverted color for a pure blue pixel is yellow, as shown in Figure 9 and explained on the basis of the arithmetic.

We could go through a similar argument to determine the colors resulting from inverting pure red and pure green. The answers, of course, would be aqua for red and fuchsia for green.

A more difficult question

What colors are produced by inverting pixels that are not pure red, green, or blue, but rather consist of weighted mixtures of red, green, and blue?

The answer to this question requires a bit of an extrapolation on our part. First, let's establish the colors that result from mixing equal amounts of the three primary colors in pairs.

- red + green = yellow *(bottom right in Figure 9)*
- red + blue = fuchsia *(bottom center in Figure 9)*
- green + blue = aqua *(bottom left in Figure 9)*

A simple color wheel

Now let's construct a simple color wheel. Draw a circle and mark three points on the circle at 0 degrees, 120 degrees, and 240 degrees. Label the first point red, the second point green, and the third point blue.

Now mark three points on the circle half way between the three points described above. Label each of these points with the color that results from mixing equal quantities of the colors identified with that point's neighbors. For example, the point half way between red and green would be labeled yellow. The point half way between green and blue would be labeled aqua, and the point half way between blue and red would be labeled fuchsia.

Look across to the opposite side

Now note the color that is on the opposite side of the circle from each of the primary colors. Aqua is opposite of red. Fuchsia is opposite of green, and yellow is opposite of blue. Comparing this with the colors shown in Figure 9, we see that the color that results from inverting one of the primary colors on the circle is the color that appears on the opposite side of the color wheel.

A reversible process

Earlier I told you that the inversion process is reversible. For example, if we have a full-intensity yellow pixel, the color values for that pixel will be:

- R = 255

- G = 255
- B = 0

If we invert the colors for that pixel, the result will be:

- R' = 255 - 255 = 0
- G' = 255 - 255 = 0
- B' = 255 - 0 = 255

Thus, the color of the inverted yellow pixel is blue, which is the color that is opposite yellow on the circle.

General conclusion

In general, we can conclude that if we invert a pixel whose color corresponds to a color at a point on the color wheel, *(such as the color wheel shown in Figure 11),* the color of the inverted pixel will match the color at the corresponding point on the opposite side of the color wheel.

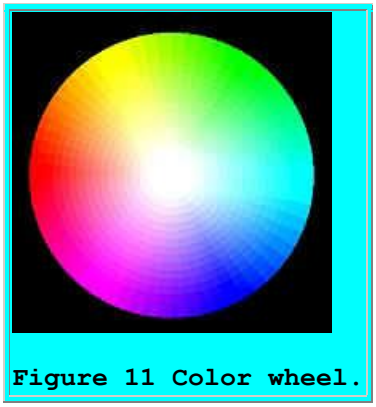

Figure 11 Color wheel.

Experimental confirmation

We can demonstrate this experimentally by inverting the image of the color wheel without performing any color filtering. The result of such an inversion is shown in the bottom half of Figure 12. Once again, the original image of the color wheel is shown at the top, and the inverted image of the color wheel is shown at the bottom.

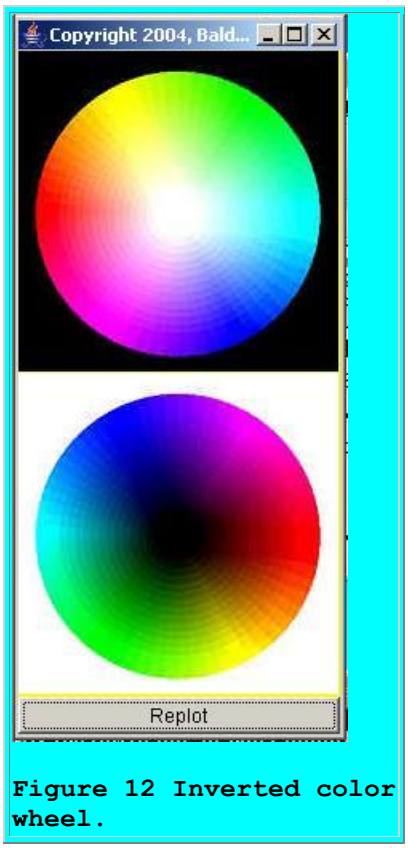

Figure 12 Inverted color wheel.

As you can see in Figure 12, each of the colors in the original image moved to the opposite side of the wheel when the color wheel was inverted.

Also, you can see from Figure 12 that white pixels turn into black pixels and black pixels turn into white pixels when they are inverted. You should be able to explain that by considering the color values for black and white pixels along with the inversion formula.

Another exercise

Another exercise might be useful. It might be possible to use the color wheel in Figure 11 to explain what happened to the colors when the starfish image was inverted in Figure 5. Pick a point on the starfish in the original image in Figure 5 and note the color of that point. Then find a point on the color wheel of Figure 11 whose color matches that point. Then find the corresponding point on the opposite side of the color wheel. The color of that point should match the color of the corresponding point on the inverted starfish image at the bottom of Figure 5.

May not have found the matching point

A potential problem here is that you may not be able to find a point on the color wheel that matches the color of a point on the starfish. That is because any individual pixel on the starfish can take on any one of 16,777,216 different colors. The colors shown on the color wheel are a small subset of that total and may not include the color of a specific point on the starfish.

Difficulty of displaying 3-dimensional data

The problem that we have here is the classic problem of trying to represent a three-dimensional entity in a two-dimensional display medium. Pixel color is a three-dimensional entity, with the dimensions being red, green, and blue. Any of the three color values belonging to a pixel can take on any one of 256 different values. It is very difficult to represent that on a flat two-dimensional screen, and a color wheel is just one of many schemes that have devised in an attempt to do so.

Could display as a cube

One way to represent these 16,777,216 colors is as a large cube having eight corners and six faces. Consider the large cube to be made up of 16,777,216 small cubes, each being a different color. Arrange the small cubes so as to form the large cube with 256 cubes *(colors)* along each edge. Thus, each face is a square with 256 small cubes along each side.

Arrange the small cubes so that the colors of the cubes at the corners on one face are black, blue, green, and aqua as shown in the top half of Figure 13. Arrange the remaining cubes on that face to contain the same colors in the same order as that shown in the top half of Figure 13.

Figure 13 Colors on one face at

top. Inverse colors at bottom.

(The colors in the bottom half of Figure 13 are the inverse of the colors shown in the top half.)

The opposite face

Arrange the small cubes such that the diagonal corners on the opposite face are set to white, yellow, red, and fuchsia as shown in the top half of Figure 14. Recall that these colors are the inverse of black, blue, green, and aqua. Arrange additional small cubes such that the colors on that face progress in an orderly manner between the colors at the corners as shown in the top half of Figure 14.

Figure 14 Colors on opposite face at top. Inverse colors at bottom.

Inverse colors

Each of the colors in the top half of Figure 14 is the inverse of the color at the diagonally opposite location on the face shown in Figure 13. For example, the yellow hues near the bottom left corner of Figure 14 are the inverse of the blues hues near the upper right corner in Figure 13.

(Also, the colors in the bottom half of Figure 14 are the inverse of the colors at the corresponding locations in the upper half of Figure 14.)

Can't show all 16,777,216 colors

In order for me to show you all 16,777,216 colors, I would have to display 128 panels like those shown in Figures 13 and 14. Each panel would represent two slices cut through the cube parallel to the two faces shown in Figures 13 and 14.

(The top half of the panel would represent one slice and the bottom half would represent the other slice.)

Each slice would represent the colors produced by combining a different value for red with all possible combinations of the values for green and blue. Obviously, it would be impractical for me to attempt to display 128 such panels in this lesson.

(Because each panel shows the raw colors at the top and the inverse colors at the bottom, only 128 such panels would be required. If only the raw colors were shown in each panel, 256 panels would be required to show all 16,777,216 colors.)

Two slices from inside the cube

The top half of Figure 15 shows a slice through the cube for a red value of 50 combined with all possible values for green and blue. The bottom half shows a slice for the inverse red value given by (255 - 50) or 205.

(Once again, the colors in the bottom half of Figure 15 are the inverse of the colors in the top half of Figure 15.)

Figure 15 A single slice through the cube for a red value of 50.

You can generate the colors yourself

Since it is impractical for me to show you all 16,777,216 colors and their inverse, I am going to do the next best thing. Listing 9 contains the program named **ImgMod27** that I used to produce the output shown in Figures 13, 14, and 15. You can compile this program and run it yourself for any value of red from 0 to 255. Just enter the red value as a command-line parameter.

The top half of the output produced by the program displays the 65,536 colors represented by a single slice through the cube parallel to the faces shown in Figures 13 and 14. The bottom half of the output in each case represents the inverse of the colors shown in the top half.

Most colors don't have names

Most of the different colors don't have names, and even if they all did have names, most of us wouldn't have them all memorized. Therefore, it is impossible for me to describe in a general sense the color that will be produced by inverting a pixel having one of the 16,777,216 possible colors.

Contribution of red, green, and blue

By doing a little arithmetic, I can describe the inverse color numerically by indicating the contribution of red, green, and blue, but most of us would probably have difficulty seeing the color in our mind's eye even if we knew the contribution of red, green, and blue.

The colors that result from some combinations of red, green, and blue are intuitive, and others are not. For example, I have no difficulty picturing that red plus blue produces fuchsia, and I have no difficulty picturing that green plus blue produces aqua. However, I am unable to picture that red plus green produces yellow. That seems completely counter-intuitive to me. I don't see anything in yellow that seems to derive from either red or green.

Of course, things get even more difficult when we start thinking about mixtures of different contributions of all three of the primary colors.

Back to experimentation

So, that brings us back to experimentation. The program in Listing 9 can be used to produce any of the 16,777,216 colors in groups of 65,536 colors, along with the inverse of each color in the group. Perhaps you can experiment with this program to produce the color that matches a point on the starfish at the top of Figure 5. If so, the inverse color shown in your output will match the color shown in the corresponding point on the starfish at the bottom of Figure 5.

And that is probably more than you ever wanted to hear about color inversion.

The remaining code

Now back to the main program named **ImgMod15**. The remaining code in the program is shown in Figure 7.

(Note that the boldface code in Listing 7 is inside a comment block.)

```
          /*Compile the following
block of code
             instead to invert before
filtering.
          temp3D[row][col][1] = (255 -
             temp3D[row][col][1]) *
redScale/100;
          temp3D[row][col][2] = (255 -
            temp3D[row][col][2]) *
greenScale/100;
          temp3D[row][col][3] = (255 -
              temp3D[row][col][3]) *
blueScale/100;
          */ end comment block
        }//end else
      }//end inner loop
```

```
    }//end outer loop
    //Return the modified array of
image data.
    return temp3D;
  }//end processImg
}//end class ImgMod15

Listing 7
```

As I mentioned earlier, the boldface code in Listing 6 filters *(scales)* the pixel first and then inverts the pixel. In some cases, it might be useful to reverse this process by replacing the boldface code in Listing 6 with the boldface code in Listing 7. This code inverts the color of the pixel first and then applies the filter. If you filter and you also invert, the order in which you perform these two operations can be significant with respect to the outcome.

The remaining code in Listing 7 signals the end of the **processImg** method and the end of the **ImgMod15** class.

Communication between the Programs

In case you are interested in the details, this section describes how the program named **ImgMod02a** communicates with the image-processing program. If you aren't interested in this much detail, just skip to the section entitled Run the Program.

Instantiate an image-processing object

During execution, the program named **ImgMod02a** 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 **ImgMod02a**:

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

All that the **ImgMod02a** 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 **ImgMod02a** 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 processed pixel data, which is displayed as an image below the original image as shown in Figure 1.

Run the Programs

I encourage you to copy, compile, and run the programs named **ImgMod15** and **ImgMod27** provided in this lesson. Experiment with them, making changes and observing the results of your changes.

Process a variety of images

Download a variety of images from the web and process those images with the program named **ImgMod15**.

> *(Be careful of transparent pixels when processing images that you have downloaded from the web. Because of the quality of the data involved, you will probably get better results from jpg images than from gif images. Remember, you will also need to copy the program named* ***ImgMod02a*** *and the interface named* ***ImgIntfc02*** *from the earlier lessons entitled Processing Image Pixels Using Java: Controlling Contrast and Brightness and Processing Image Pixels using Java, Getting Started.)*

View a large number of different colors

Compute and observe the colors and their inverse for various slices through the color cube as provided by the program named **ImgMod27**.

Change the order of filtering and inversion

Run some experiments to determine the difference in results for various images based on filtering before inverting and on inverting before filtering.

> *(Of course, if you don't filter, it won't matter which approach you use.)*

Write an advanced filter program

Write an advanced version of the program that applies color filtering by allowing you to control both the location and the width of the distribution for each of the three colors separately. You can get some ideas on how to do this from the program entitled Processing Image Pixels Using Java: Controlling Contrast and Brightness.

Replicate the results

To replicate the results shown in this lesson, right-click and download the jpg image files in Figures 17, 18, and 19 below.

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

Figures 17, 18, and 19 contain the jpg images that were used to produce the results shown in this lesson. You should be able to right-click on the images to download and save them locally. Then you should be able to replicate the results shown in this lesson.

Figure 17

Figure 18

Figure 19

Summary

In this lesson, I showed you how to write a Java program that can be used to:

- Control color intensity
- Apply color filtering
- Apply color inversion

I provided several examples of these capabilities. In addition, I explained some of the theory behind color inversion and showed you how to relate the colors on original and inverted pixels to points on a color wheel as well as pixels in a color cube.

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.
- 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 Listings

Complete listings of the programs discussed in this lesson are provided in Listings 8 and 9.

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 ImgMod15.java.java
Copyright 2005, R.G.Baldwin

This program illustrates color intensity control,
color filtering, and color inversion.

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

java ImgMod02 ImgMod15 gifFileName

The program places two GUIs on the screen. One
GUI displays the "before" and "after" images of
an image that is subjected to color intensity
cotrol, color filtering, and color inversion. The
image at the top is the "before" image.  The
image at the bottom is the "after" image.

The other GUI provides instructions and
components by which the user can control the
processing of the image.  A check box appears
near the top of this GUI.  If the user checks
the check box, color filtering is performed
before color inversion takes place.  If the
check box is not checked, no color inversion is
performed.

This GUI also provides three sliders that make
it possible for the user to control color
intensity and filtering.  Each slider controls
the intensity of a single color.  The intensity
control ranges from 0% to 100% of the original
intensity value for each pixel.

If all three sliders are adjusted to the same
percentage value and the replot button is
pressed, the overall intensity of the image is
modified with no change in the relative
contribution of each color. This makes it
possible to control the overall intensity of the
image from very dark (black) to the maximum
intensity supported by the image.
```

```
If the three sliders are adjusted to different
percentage values and the replot button is
pressed, color filtering occurs. By this, I mean
that the intensity of one color is changed
relative to the intensity of the other colors.
This makes it possible, for example to adjust the
"warmth" of the image by emphasizing red over
blue, or to make the image "cooler" by
emphasizing blue over red.  It is also possible
to totally isolate and view the contributions of
red, green, and blue to the overall image.

As written, the program applies color filtering
before color inversion.  Sample code is also
provided that can be used to modify the program
to cause it to provide color inversion before
color filtering.  There is a significant
difference in the results produced by these two
different approaches.

As an interesting side note, Microsoft Word and
Microsoft FrontPage appear to use color
inversion to change the colors in images that
have been selected for editing.

This program illustrates the modification of
the pixels in an image.  It works best with
an image file that contains no transparent areas.

The pixel modifications performed in this program
have no impact on transparent pixels.  If you
don't see what you expect, it may be because your
image contains transparent pixels.

Tested using JDK 1.5.0 and WinXP
**********************************************/
import java.awt.*;
import javax.swing.*;

class ImgMod15 extends Frame
                           implements ImgIntfc02{

  //GUI components used to control color
  // filtering and color inversion.
  JSlider redSlider;
  JSlider greenSlider;
  JSlider blueSlider;
  JCheckBox checkBox;

  //Constructor must take no parameters
  ImgMod15(){
    //Create and display the user-input GUI.
    setLayout(new FlowLayout());

    //Provide user instructions at the top of the
```

```
    // GUI.
    add(new Label(
            "Adjust Color Sliders and Replot."));

    //Provide a check box that is used to request
    // color inversion.
    checkBox = new JCheckBox();
    add(new Label("Check to Invert Colors"));
    add(checkBox);

    //Create three sliders each with a range of
    // 0 to 100 and an initial value of 100.
    redSlider = new JSlider(0,100,100);
    add(redSlider);
    //Put a label under the slider.
    add(new Label("Red"));

    greenSlider = new JSlider(0,100,100);
    add(greenSlider);
    add(new Label("Green"));

    blueSlider = new JSlider(0,100,100);
    add(blueSlider);
    add(new Label("Blue"));

    //Put numeric labels and tick marks on the
    // sliders.
    redSlider.setMajorTickSpacing(20);
    redSlider.setMinorTickSpacing(5);
    redSlider.setPaintTicks(true);
    redSlider.setPaintLabels(true);

    greenSlider.setMajorTickSpacing(20);
    greenSlider.setMinorTickSpacing(5);
    greenSlider.setPaintTicks(true);
    greenSlider.setPaintLabels(true);

    blueSlider.setMajorTickSpacing(20);
    blueSlider.setMinorTickSpacing(5);
    blueSlider.setPaintTicks(true);
    blueSlider.setPaintLabels(true);

    setTitle("Copyright 2005, Baldwin");
    setBounds(400,0,220,330);
    setVisible(true);
  }//end constructor

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

    //Make a working copy of the 3D array to
    // avoid making permanent changes to the
```

```
    // raw 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 the current values of the three
    // sliders.  This information will be used to
    // scale the red, green, and blue pixel
    // values to new values ranging from 0% to
    // 100% of the original values in order to
    // implement color filtering.
    int redScale = redSlider.getValue();
    int greenScale = greenSlider.getValue();
    int blueScale = blueSlider.getValue();

    //Process each pixel value to apply color
    // filtering either with, or without color
    // inversion depending on the state of the
    // check box.
    for(int row = 0;row < imgRows;row++){
      for(int col = 0;col < imgCols;col++){
        if(!checkBox.isSelected()){
          //Apply color filtering but no color
          // inversion
          temp3D[row][col][1] =
              temp3D[row][col][1] * redScale/100;
          temp3D[row][col][2] =
            temp3D[row][col][2] * greenScale/100;
          temp3D[row][col][3] =
             temp3D[row][col][3] * blueScale/100;
        }else{
          //Apply color filtering with inversion.
          // Compile the following block of code
          // to filter before inverting.
          temp3D[row][col][1] = 255 -
              temp3D[row][col][1] * redScale/100;
          temp3D[row][col][2] = 255 -
            temp3D[row][col][2] * greenScale/100;
          temp3D[row][col][3] = 255 -
             temp3D[row][col][3] * blueScale/100;

          /*Compile the following block of code
             instead to invert before filtering.
          temp3D[row][col][1] = (255 -
             temp3D[row][col][1]) * redScale/100;
```

```
          temp3D[row][col][2] = (255 -
           temp3D[row][col][2]) * greenScale/100;
          temp3D[row][col][3] = (255 -
            temp3D[row][col][3]) * blueScale/100;
          */
        }//end else
      }//end inner loop
    }//end outer loop
    //Return the modified array of image data.
    return temp3D;
  }//end processImg
}//end class ImgMod15
```

Listing 8

```
/*File ImgMod27.java Copyright 2005, R.G.Baldwin
Creates a Frame containing 65536 colors at the
top and the inverse of those 65536 colors at the
bottom for a total of 131072 different colors.

Enter a color for red between 0 and 255 at the
command line.  Program displays 65536 colors
based on that color for red combined with all
possible combinations of green and blue in the
top half of the Frame.

The 131072 colors are surrounded by a thin
yellow border, which in turn is surrounded by
a wider gray border, all inside a Frame.

Tested using JDK 1.5 under WinXP.
**********************************************/

import java.awt.*;
import java.awt.event.*;

public class ImgMod27 extends Frame{

  static int red = 0;

  public static void main(String[] args){
    //Get input value for red.
    if(args.length == 1){
      red = Integer.parseInt(args[0]);
    }else{
      System.out.println(
                "Usage: java ImgMod27 redValue\n"
              + "Using red = 0");
    }//end else

    //Confirm that red is within range.
    if((red < 0) || (red > 255)){
```

```
      System.out.println(
                  "Red must be >= 0 and <= 255\n"
                + "Terminating");
      System.exit(0);
    }//end if
    new ImgMod27();
  }//end main

  ImgMod27(){//constructor
    int panelWidth = 256 + 4;
    int panelHeight = 512 + 4;
    setVisible(true);
    setBackground(Color.gray);
    Insets insets = this.getInsets();
    setSize(panelWidth + insets.left
                              + insets.right + 8,
            panelHeight + insets.top
                            + insets.bottom + 8);
    setLayout(null);

    MyPanel panel = new MyPanel();
    panel.setBounds(insets.left + 4,
          insets.top + 4,panelWidth,panelHeight);
    panel.setBackground(Color.YELLOW);
    add(panel);
    setTitle("Copyright 2005, Baldwin");
    //=======================================//

    //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
  class MyPanel extends Panel{
    int green = 0;
    int blue = 0;
    int xcoor = 1;
    int ycoor = 1;
    public void paint(Graphics g){
      //Compute and display colors at top of
      // Frame.
      for(green = 0;green < 256;green++){
        for(blue = 0;blue < 256;blue++){
          g.setColor(new Color(red,green,blue));
          g.drawOval(xcoor++,ycoor,1,1);
          if(xcoor > 256){
            xcoor = 1;
            ycoor++;
```

```
          }//end if
        }//end inner loop
      }//end outer loop

      //Now compute and display inverted colors
      // at bottom of Frame.
      for(green = 0;green < 256;green++){
        for(blue = 0;blue < 256;blue++){
          g.setColor(new Color(255-red,
                               255-green,255-blue));
          g.drawOval(xcoor++,ycoor,1,1);
          if(xcoor > 256){
            xcoor = 1;
            ycoor++;
          }//end if
        }//end inner loop
      }//end outer loop
    }//end overridden paint
  }//end inner class MyPanel
}//end class ImgMod27
```

Listing 9

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 have 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

Keywords
java image pixel color intensity filtering inversion framework GUI transparent slider experiment wheel cube

-end-