Processing Image Pixels using Java, Creating a Spotlight

Baldwin shows you how to control the brightness of pixels in an image. As an example, he shows you how to create a spotlight that lights up the center of an image leaving the outer edges dark.

Published: October 19, 2004
By Richard G. Baldwin

Java Programming, Notes # 402

Preface

Second in a series

This lesson is the second lesson in a series that will 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. Because this lesson builds upon that earlier lesson, you will need to understand the code in the previous lesson before the code in this lesson will make much sense.

Not a lesson on JAI

If you arrived at this lesson while searching for instructions on how to use the Java Advanced Imaging *(JAI)* API, you are at the wrong place. While you are certainly welcome to be here, that is not the purpose of the lessons in this series. The purpose of this series is to teach you how to implement many of the algorithms that are commonly used to create special effects with images by working directly with the pixels.

A framework or driver program

The previous 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 for easy comparison in a *before* and *after* sense *(see Figure 1)*

The program named **ImgMod02** is designed to be used as a framework or driver that controls the execution of a second program. The second program actually processes the pixels.

By using **ImgMod02** as a driver, you can concentrate on writing and executing image-processing algorithms without having to worry about many of the details such as reading image files, displaying images, etc.

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 run the program that I will provide in this lesson.

An image-processing program

The earlier lesson entitled Processing Image Pixels using Java, Getting Started also provided and explained a very simple image-processing program. That program was designed to introduce you to the concept of modifying an image by directly modifying the pixels that belong to the image. I will provide and explain a more substantive image-processing program in this lesson.

The image-processing program that I will explain in this lesson will show you how to create a spotlight that lights up the center of an image while leaving the outer edges dark.

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.

Creating a spotlight in an image

Figure 1 shows an example of the output produced by the program that I will explain in this lesson. The special effect illustrated by Figure 1 begins with a picture of a starfish taken in a well-illuminated aquarium and converts it to what looks like a picture taken by a SCUBA diver deep underwater.

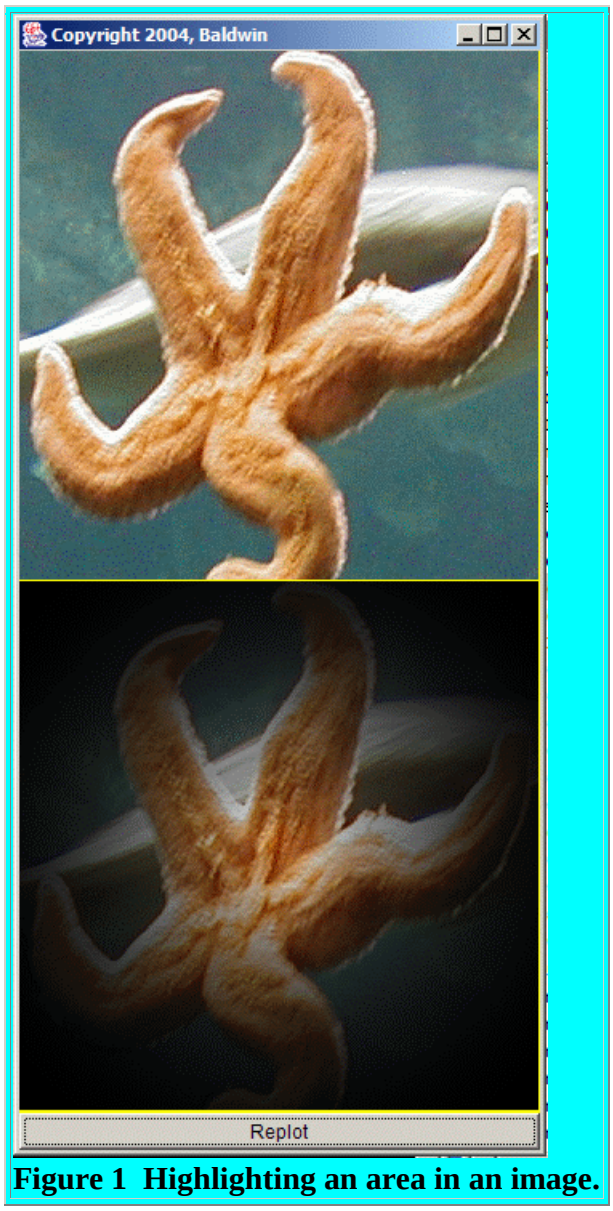

Figure 1 Highlighting an area in an image.

In Figure 1, as in all of the graphic output produced by the driver program named **ImgMod02**, the original image is shown at the top and the modified image is shown at the bottom.

The image-processing program illustrated by Figure 1 allows the user to interactively control the degree to which the light is concentrated in the center of the picture. Depending on user input, the illumination can range from being concentrated in a very small area in the center of the image to being spread throughout the image.

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 previous 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 in Java *(and in most modern computer systems for that matter).*

The 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 previous lesson also explained that the lessons in this series are not concerned about file formats. Rather, the lessons are concerned with what to do with the pixels once they have been extracted from the file.

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 previous lesson.

Will concentrate on the three-dimensional array of type int

This lesson and future lessons in this series will concentrate on writing image-processing programs that implement a variety of image-processing algorithms. As a result, these lessons will concentrate on receiving and modifying the contents of the three-dimensional array containing pixel data as type **int**. The image-processing programs will receive raw pixel data in the form of a three-dimensional array, and will return modified pixel data in the form of a three-dimensional array.

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.

A quick review of fundamentals

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

Mixing the primary colors red, green, and blue

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

The amounts of each of the three primary colors that are added are specified by the values stored in the three color values for the pixel.

The range of a color

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

Black and white pixels

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

The bottom line on color

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

What about transparency?

The issue of transparency is somewhat more difficult to explain. I explained this concept using an analogy in the previous lesson entitled Processing Image Pixels using Java, Getting Started. I won't repeat that analogy here. Rather, I will simply refer you back to the earlier lesson.

The bottom line on transparency is as follows. The alpha byte also has 256 possible values ranging from 0 to 255. If the value is zero, the pixel is completely transparent regardless of the values of the three color bytes.

If the value is 255, the pixel is completely opaque with the color of the pixel being determined exclusively by the values stored in the three color bytes.

If the value of the alpha byte is somewhere between 0 and 255, the pixel is partially transparent. The overall color of that pixel depends on the color that was previously established for that location plus the individual values of the three color bytes.

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 actually involves two programs and one interface. One of the programs is the program named **ImgMod02** that I provided and explained in the previous lesson entitled Processing Image Pixels using Java, Getting Started. The interface is named **ImgIntfc02**. I also provided and explained that interface in the previous lesson. I will refer you back to that lesson for access to that material.

In this lesson, I will present and explain a new Java program named **ImgMod11**. This program, when run under control of the program named **ImgMod02**, will produce an output similar to the pair of images shown in Figure 1.

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

The processImg method

The program named **ImgMod11**, *(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 causes the original image and the modified image to be displayed in a frame with the original image above the modified image *(see Figures 1 for an example of the display format).*

Usage information for ImgMod02 and ImgMod11

To use the program named **ImgMod02** to drive the program named ImgMod11, enter the following at the command line:

```
java ImgMod02 ImgMod11 ImagePathAndFileName
```

Image file must be provided by the user

The user must provide the image file. It can be a ***gif*** file or a ***jpg*** file. Other file types may also be compatible as well. The image file doesn't have to be in the current directory if a path to the file is specified on the command line. If the program is unable to load the image file within ten seconds, it will abort with an error message.

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

Image display format

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

A **Replot** button appears at the bottom of the frame. If the user clicks the **Replot** button, the **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 **ImgMod11** provides a GUI for data input. This makes it possible for the user to modify the behavior of the image-processing method each time it is run.

Discussion and Sample Code

The processImg method

As mentioned earlier, the image-processing program must implement the interface named **ImgIntfc02**. A listing of that interface was provided in the previous 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 columns of pixels in the image.

It's best to make and modify a copy

The image-processing class must define the method named **processImg** with the signature given above. The **processImg** method receives a three-dimensional array containing pixel data for the original image. Normally the 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 as expected.

> *(It is possible, however, to create some interesting visual effects by taking advantage of such truncation. You might want to play around with this.)*

Dealing with out-of-range values

There are at least two ways to deal with out-of-range values. 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 map the values from the most negative to the most positive into the range from 0 to 255. With this approach, all the pixel values would be modified the same way such that the minimum value contained in all the pixel color values would be 0 and the maximum value would be 255.

There are many possible variations on the latter approach. There is no one approach that is *the right* approach for all situations.

Instantiate an image-processing object

The program named **ImgMod02** reaches a point where it has captured the pixel data from the original image in a three-dimensional array of type **int** suitable for processing. Then it invokes the **newInstance** method of the class named **Class**, as shown in Listing 1, to instantiate an object of the image-processing class.

> *(At this point, the name of the image-processing class is stored as a **String** in the variable named **theProcessingClass**. The **String** was either received as a command-line parameter, or was specified as a default value.)*

```
        try{
          imageProcessingObject =
            (ImgIntfc02)Class.forName(

theProcessingClass).newInstance();
```

Listing 1

The use of the class named Class

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

As explained in the previous lesson entitled Processing Image Pixels using Java, Getting Started, this approach does not support the use of parameterized constructors.

Fire an ActionEvent

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 is to invoke the **processImg** method on the image-processing object passing the pixel data and other appropriate information as parameters.

The **ImgMod02** program executes the statement in Listing 2 to invoke the **processImg** method on the image-processing object.

```
          Toolkit.getDefaultToolkit().

getSystemEventQueue().postEvent(
                   new ActionEvent(
                     replotButton,

ActionEvent.ACTION_PERFORMED,
                      "Replot"));
```

Listing 2

Post a counterfeit ActionEvent to the system event queue

Listing 2 posts a counterfeit **ActionEvent** to the system event queue and attributes the event to the **Replot** button. The result is exactly the same as if the user had clicked the **Replot** button 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.

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

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 ImgMod11

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

```
java ImgMod02 ImgMod11 ImagePathAndFileName
```

Purpose of the program

This program illustrates control of the light intensity in the image. The effect is as though a spotlight is shining on the object in the center of the image. The center of the image is illuminated while the outer edges of the image are dark, as shown in Figure 1.

User input

A small GUI with a text field allows the user to control the radius of the illuminated area. At startup, the text field contains a default value of 0. To change the radius of the illuminated area, type a positive integer value into the text field and press the **Replot** button at the bottom of the image. The larger the integer value, the smaller will be the radius of the illuminated area.

(The bottom image in Figure 1 resulted from a user input value of 1 in the text field.)

For values of around 10 and above, the illuminated area will be reduced to approximately the size of a single pixel.

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.

The program was tested using J2SE 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 14 near the end of the lesson.

The class definition begins in Listing 3. Note that this class implements **ImgIntfc02**. This is a requirement to make this class compatible with the driver program named **ImgMod02**.

```
class ImgMod11 extends Frame
                            implements
ImgIntfc02{

  int loops;//User input to control
lighting
  String inputData;//Obtained via the
TextField
  TextField input;//User input field

  ImgMod11(){//constructor
    //Create a user input frame.
    setLayout(new FlowLayout());

    Label instructions = new Label(
                        "Type an int and
replot.");
    add(instructions);

    input = new TextField("0",5);
    add(input);

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

```
Listing 3
```

The class also extends **Frame**. This is because an object of the class serves as a GUI containing a **TextField** that allows for user input.

Listing 3 shows three instance variables and the constructor. The instance variables will be discussed later in conjunction with the discussion of the code that uses them.

The constructor

The constructor displays a small **Frame** on the screen with a single **TextField** object. The purpose of the text field is to allow the user to enter a value that controls the radius of the illuminated area.

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 code in Listing 3 is straightforward and shouldn't require further explanation.

The processImg method

The class defines the **processImg** method that is declared in the interface named **ImgIntfc02**.

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

The **processImg** method begins in Listing 4.

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

    //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

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

Listing 4
```

A working copy of the array

The **processImg** method begins by making a working copy of the incoming three-dimensional array in order to avoid having to make changes to the original array of pixel data.

Then the method displays the width and height of the image in pixels as information to the user.

Get user input

Listing 5 gets the **String** value currently in the **TextField** and converts that **String** to a value of type **int**.

```
    loops =
Integer.parseInt(input.getText());

Listing 5
```

The first time that the **processImg** method is called, it get the value 0 that was put into the **TextField** through initialization when the **TextField** object was instantiated. After that, the value retrieved will be the value typed into the **TextField** by the user.

Note that there is no requirement for the user to press the **Enter** key after typing a new value into the **TextField**. However, pressing the **Enter** key won't do any harm.

Also note that the code in Listing 5 will throw an exception if the contents of the **TextField** cannot be converted to a value of type **int**.

Calculate the center point

The code in Listing 6 calculates the coordinate value of the pixel at the center of the image.

```
    int centerWidth = imgCols/2;
    int centerHeight = imgRows/2;
```

Listing 6

This is the point that will be the center of the illuminated area in the modified image.

Get the distance from a corner

Listing 7 calculates the distance from a corner to the center pixel in the image. This will be used as a base for calculating the relative distance from any pixel to the center.

```
    double maxDistance = Math.sqrt(
centerWidth*centerWidth +
centerHeight*centerHeight);
```

Listing 7

The distance from the corner is calculated by calculating the length of the hypotenuse of a right triangle formed by the distances from the center to the side and to the top of the image.

Control pixel intensity

The effect that you see in Figure 1 is achieved by decreasing the intensity of every pixel in the image except for the pixel at the center.

For the initial value of 0 in the **TextField**, the intensity of each pixel decreases linearly with the distance of the pixel from the center.

When the user types a non-zero positive integer value into the **TextField** and presses the **Replot** button, the intensity of each pixel decreases more rapidly with the distance of the pixel from the center.

> *(For example, for a value of 1 in the **TextField**, the intensity decreases as the square of the distance of the pixel from the center.)*

A pair of nested for loops

Listing 8 shows the beginning of a pair of nested **for** loops that are used to modify the intensity of each pixel on the basis of the location of the pixel in the image.

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

        double horiz = centerWidth -
col;
        double vert = centerHeight -
row;
```

Listing 8

Get components of displacement vector

Listing 8 also contains the code to calculate the horizontal and vertical components of a displacement vector that extends from a pixel to the center of the image.

Get length of displacement vector

Listing 9 calculates the absolute value of the length of the displacement vector as the length of the hypotenuse of a right triangle formed by the horizontal and vertical components of the displacement vector.

```
        double distance =
Math.sqrt(horiz*horiz
                                    +
vert*vert);
```

Listing 9

The absolute value of the displacement vector is the scalar distance from the pixel to the center of the image.

Calculate the scale factor

Listing 10 shows the beginning of an **if** statement that is executed only if the distance to the pixel from the center is greater than zero.

> *(The body of the **if** statement will be executed for all pixels other than the single pixel at the center of the image. Therefore, the intensity of the center pixel should never change, no matter what positive value is entered into the **TextField**, provided that value can be converted to a positive **int** value.)*

```
        if((int)distance > 0){
          double scale =
                        1.0 -
distance/maxDistance;

Listing 10
```

Listing 10 calculates a scale factor that is used later to scale each of the three color values belonging to an individual pixel.

For small distances, the scale factor is very close to unity. For a distance equal to the distance from a corner to the center, the scale factor is zero. Thus, the intensity of the pixels near the center will be nearly as great as the intensity of the same pixels in the original image. The intensity of pixels in the corners will be reduced to zero.

Modify the scale factor based on user input

The **for** loop in Listing 11 modifies the scale factor based on the contents of the **TextField**.

```
          for(int cnt = 0;cnt <
loops;cnt++){
            scale = scale*scale;
          }//end for loop

Listing 11
```

For the initial default value of 0 in the **TextField**, the body of the **for** loop is not executed. Thus, the value of the scale factor isn't modified for this case. The scale factor decreases linearly with the distance of the pixel from the center.

For a value of 1 in the **TextField**, the value of the scale factor decreases as the square of the distance of the pixel from the center. For a value of 2, the scale factor decreases as the cube of the distance, etc.

Apply the scale factor to each pixel

Still inside the pair of nested **for** loops that began in Listing 8, the code in Listing 12 multiplies the red, green, and blue values of the pixel by the scale factor.

(No change is made to the alpha or transparency value.)

```
          temp3D[row][col][1] =
              (int)(scale *
temp3D[row][col][1]);
          temp3D[row][col][2] =
```

```
              (int)(scale *
temp3D[row][col][2]);
          temp3D[row][col][3] =
              (int)(scale *
temp3D[row][col][3]);

Listing 12
```

As discussed earlier, the color values of the pixels at the corners are reduced to zero. When all three pixels have a value of zero, the overall color of the pixel will be black. Similarly, each of the color values for a pixel between the corners and the center are reduced by the same scale factor. This causes the portion of the image attributable to those pixels to appear to be more poorly illuminated.

Return modified pixel data

Listing 13 signals the end of the **if** statement that began in Listing 10. Listing 13 also signals the end of the nested **for** loops that began in Listing 8.

Listing 13 returns a reference to the three-dimensional array object that contains the modified pixel data. This is the data that will be displayed by the program named **ImgMod02**.

```
        }//end if
      }//end for loop on col
    }//end for loop on row

    return temp3D;
  }//end processImg

}//end class ImgMod11

Listing 13
```

Listing 13 also signals the end of the **ImgMod11** class and the end of the program.

That's all there is to it

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

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

Some cautions

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

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

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

Value ranges

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

Prior to display, each of the four pixel values of type **int** will be converted to eight bits by simply discarding all but the least significant eight bits in each **int** element in the three-dimensional array. This is not the same as clipping the values at 0 and 255, and will probably lead to unexpected results.

Run the Program

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

Variable transparency

If you were to modify the alpha value along with the color values, the pixels would become more transparent as they become darker. This produces an interesting effect.

Investigation by individual color

The code in Listing 14 contains some statements *(disabled by comments)* that you can use to turn colors off and on. This allows you to see the effect of the algorithm on any combination of the three colors. Thus, you can see the impact of applying the algorithm to each individual color among red, green, and blue.

Moving the center of the illuminated area

Another interesting programming exercise would be to allow the user to enter two values in the GUI with one value controlling the radius of the illuminated area, and the other value controlling the center of the illuminated area.

An even more interesting exercise would be to make it possible for the user to control the center of the illuminated area by moving the mouse around in the top image in Figure 1. There are at least two ways to do this.

The easier of the two ways would be to cause the center of the illuminated area to move to a new location each time the user clicks on that location in the top image.

A more difficult approach would be to cause the center to the illuminated area to track the location of the mouse pointer in the top image without a requirement for the user to press a mouse button.

> *(This would probably require more raw compute power than my old computer can muster, particularly for a large image.)*

Have fun and learn

Above all, have fun and learn as much as you can about modifying image pixels using Java.

A test image

You should be able to right-click on the image in Figure 2 and download and save it locally. Then you should be able to replicate the output produced in Figure 1.

Figure 2 Raw image for Figure 1

Summary

I showed you how to control the brightness of pixels in an image. As an example, I showed you how to create a spotlight that lights up the center of an image while leaving the outer edges dark.

I will have more to say about controlling brightness in the next lesson in this series where I explain how to control both brightness and contrast.

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:

- Control contrast and brightness by modifying the distribution of the color values.
- 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

A complete listing of the program discussed in this lesson is provided in Listing 14.

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 special effects built in if you prefer to write your own production image-processing programs using Java.

The programs that I will provide in this series 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 ImgMod11.java.java
Copyright 2004, R.G.Baldwin

This program is designed to be driven by the
program named ImgMod02.java.

Enter the following on the command line to run
```

```
this program:

java ImgMod02 ImgMod11 FileName

This program illustrates an image special effect
involving the control of light intensity in the
image.  The effect is as though a spotlight is
shining on the object in the center of the image.
The center of the image is well lit while the
outer edges of the image are dark.

A small GUI with a text field is provided to
allow the user to control the radius of the
lighted area.  At startup, the text field
contains a 0.  To change the radius of the
lighted area, type an integer value into the
text field and press the Replot button at the
bottom of the image.  The larger the integer
value, the smaller will be the radius of the
lighted area.  For values of around 10 and
above, the lighted area will be reduced to
approximately the size of a single pixel.

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

Tested using SDK 1.4.2 and WinXP
*********************************************/
import java.awt.*;

class ImgMod11 extends Frame
                           implements ImgIntfc02{

  int loops;//User input to control lighting
  String inputData;//Obtained via the TextField
  TextField input;//User input field

  ImgMod11(){//constructor
    //Create a user input frame.
    setLayout(new FlowLayout());

    Label instructions = new Label(
                      "Type an int and replot.");
    add(instructions);

    input = new TextField("0",5);
    add(input);

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

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

    //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

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

    //Get the user input from the text field.
    // First time through, the input value is the
    // value set into the text field when it was
    // instantiated.  After that, the input value
    // is the value typed by the user.
    loops = Integer.parseInt(input.getText());

    //Calculate the center point of the image
    int centerWidth = imgCols/2;
    int centerHeight = imgRows/2;

    //Calculate the distance from a corner to the
    // center point of the image.  This will be
    // used as a base for calculating the
    // relative distance from any pixel to the
    // center.
    double maxDistance = Math.sqrt(
                       centerWidth*centerWidth +
                       centerHeight*centerHeight);

    //Decrease the intensity of every pixel
    // except the one in the center of the image.
    // Initially the intensity will decrease
    // linearly with the distance of a pixel from
    // the center pixel.  The user can modify
    // this through input at the text field
    // later.
```

```
    //Use a nested loop to operate on each pixel.
    for(int row = 0;row < imgRows;row++){
      for(int col = 0;col < imgCols;col++){
        //Get the horizontal and vertical
        // distances from the center as the two
        // sides of a right triangle.
        double horiz = centerWidth - col;
        double vert = centerHeight - row;

        //Get the distance to the center as the
        // hypotenuse of a right triangle.
        double distance = Math.sqrt(horiz*horiz
                                    + vert*vert);
        if((int)distance > 0){
          //Calculate the scale factor to apply
          // to the three color values for the
          // pixel in order to control the
          // intensity.
          double scale =
                      1.0 - distance/maxDistance;

          //Adjust the light intensity based on
          // user input.  For a user input value
          // of 0 (the default at startup), the
          // intensity decreases linearly with
          // the distance from the center.  For a
          // user input value of 1, the intensity
          // decreases as the square of the
          // distance.  For an input value of 2,
          // the intensity decreases as the cube
          // of the distance, etc.
          for(int cnt = 0;cnt < loops;cnt++){
            scale = scale*scale;
          }//end for loop

          //Multiply each of the three color
          // values by the scale factor.  Don't
          // make any changes to the alpha value.
          temp3D[row][col][1] =
              (int)(scale * temp3D[row][col][1]);
          temp3D[row][col][2] =
              (int)(scale * temp3D[row][col][2]);
          temp3D[row][col][3] =
              (int)(scale * temp3D[row][col][3]);
/*
          //As an interesting experiment, you can
          // filter out two colors by enabling
          // these two statements and setting the
          // value of the third dimension to 1,
          // 2, or 3.  As written here, green and
          // blue will be eliminated leaving only
          // red.
          temp3D[row][col][2] = 0;
          temp3D[row][col][3] = 0;
*/
        }//end if
```

```
      }//end for loop on col
    }//end for loop on row

    return temp3D;
  }//end processImg
  //----------------------------------------//

}//end class ImgMod11
```

Listing 14

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-