Using the Java 2D BandCombineOp Filter Class to Process Images

Learn how to write programs that extract Raster objects from an image and then use the BandCombineOp image-filtering class of the Java 2D API for a variety of image-processing purposes.

Published: July 3, 2007
By Richard G. Baldwin

Java Programming Notes # 458

Preface

Part of a series

In an earlier lesson entitled "A Framework for Experimenting with Java 2D Image-Processing Filters" *(see References)*, I taught you how to write a framework program that makes it easy to use the image-filtering classes of the Java 2D API to process the pixels in an image and to display the processed image.

At the close of that lesson, I told you that future lessons would teach you how to use the following image-filtering classes from the Java 2D API:

- **LookupOp**
- **AffineTransformOp**
- **BandCombineOp**
- **ConvolveOp**
- **RescaleOp**
- **ColorConvertOp**

In several previous lessons listed in the References section, I taught you how to use the **LookupOp** and the **AffineTransformOp** image-filtering classes.

In this lesson, I will teach you how to use the **BandCombineOp** image-filtering class to perform a variety of filtering operations on images. I will also teach you how to extract and filter **Raster** objects from images. *(The use of* ***Raster*** *objects is completely new to this lesson.)*

I will teach you how to use the remaining classes from the above list in future lessons.

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 listings and figures 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.

I also recommend that you pay particular attention to the lessons listed in the References section of this document.

General Background Information

Constructing images

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

I provided a great deal of information on those topics in the earlier lesson entitled Processing Image Pixels using Java, Getting Started. Therefore, I won't repeat that information here. Rather, I will simply refer you back to the earlier lesson.

The framework program named ImgMod05

It will also be useful for you to understand the behavior of the framework program named **ImgMod05**. Therefore, I strongly recommend that you study the earlier lesson entitled "A Framework for Experimenting with Java 2D Image-Processing Filters" *(see References).*

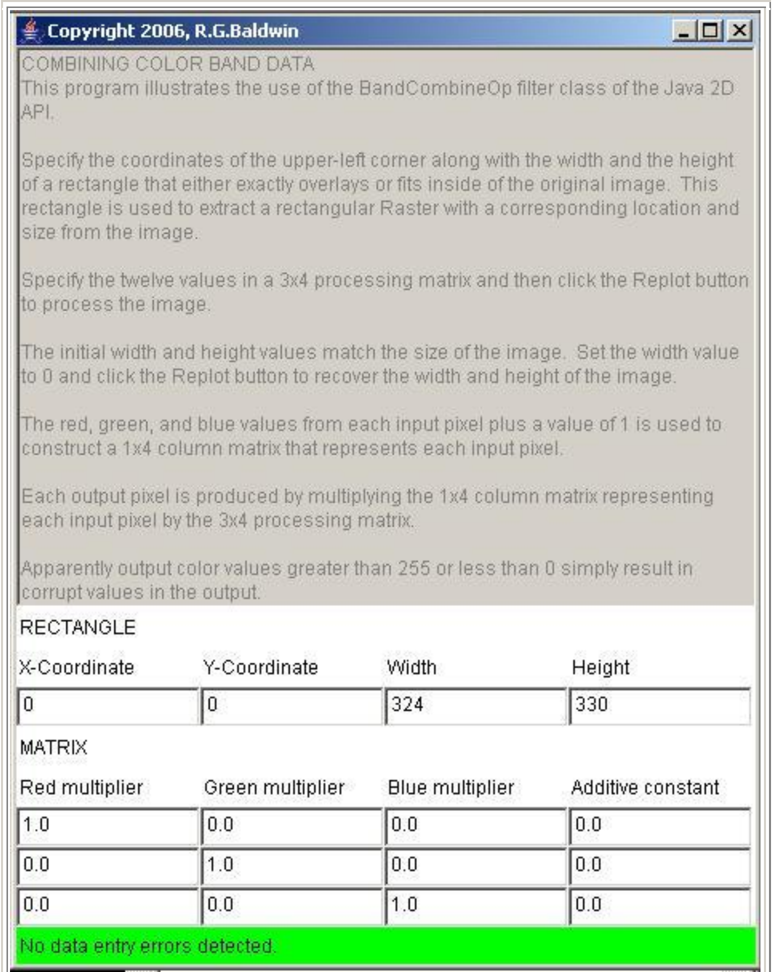
However, if you don't have the time to do that, you should take a look at the earlier lesson entitled "Using the Java 2D LookupOp Filter Class to Process Images" *(see References)*, in which I summarized the behavior of the framework program named **ImgMod05**.

Preview

In this lesson, I will present and explain an image-processing program named **ImgMod41** that is compatible with the framework program named **ImgMod05**.

The program GUI

The program GUI is shown in Figure 1.



Copyright 2006, R.G.Baldwin

COMBINING COLOR BAND DATA
This program illustrates the use of the BandCombineOp filter class of the Java 2D API.

Specify the coordinates of the upper-left corner along with the width and the height of a rectangle that either exactly overlays or fits inside of the original image. This rectangle is used to extract a rectangular Raster with a corresponding location and size from the image.

Specify the twelve values in a 3x4 processing matrix and then click the Replot button to process the image.

The initial width and height values match the size of the image. Set the width value to 0 and click the Replot button to recover the width and height of the image.

The red, green, and blue values from each input pixel plus a value of 1 is used to construct a 1x4 column matrix that represents each input pixel.

Each output pixel is produced by multiplying the 1x4 column matrix representing each input pixel by the 3x4 processing matrix.

Apparently output color values greater than 255 or less than 0 simply result in corrupt values in the output.

RECTANGLE

X-Coordinate	Y-Coordinate	Width	Height
0	0	324	330

MATRIX

Red multiplier	Green multiplier	Blue multiplier	Additive constant
1.0	0.0	0.0	0.0
0.0	1.0	0.0	0.0
0.0	0.0	1.0	0.0

No data entry errors detected.

Figure 1

In addition to providing instructions to the user, this GUI allows the user to specify the location and dimensions of a rectangular area within the original image, from which a **Raster** object will be extracted. The GUI also allows the user to specify the values in a 3x4 image-processing matrix having three rows and four columns that is used to process the raster using the **BandCombineOp** image-filtering class.

Buffered v.s. Raster objects.
I have illustrated the filtering

BufferedImage objects versus Raster objects

of BufferedImage objects in several previous lessons listed in the References section. I will have more to say about Raster objects later in this lesson. For now, suffice it to say that by converting the BufferedImage object to a Raster object, it is possible to operate on smaller rectangular areas of the image that are extracted from within the body of the entire image.

As I mentioned earlier, the use of **Raster** objects is completely new to this lesson.

Unlike some of the other image-filtering classes in the Java 2D API that can operate either on **BufferedImage** objects or on **Raster** objects, the **BandCombineOp** filter can operate only on **Raster** objects.

Image-filtering methodology

For the **BandCombineOp** class, the red, green, and blue values of each pixel are treated as a column matrix. A 1 is appended onto the end of each column matrix producing a set of four-element column matrices that represents all of the pixels in the input **Raster** object. *(Each pixel is represented by a four-element column matrix.)*

Each pixel in the output **Raster** is produced by multiplying a user-specified 3x4 image-processing matrix by the 4x1 column matrix that represents the corresponding pixel in the input **Raster**. The same 3x4 processing matrix is applied to every input pixel. *(An example of the values in such a 3x4 image-processing matrix is shown in the bottom of Figure 1.)*

This makes it possible to cause the intensity or shade of each of the three colors *(red, green, and blue)* in each pixel of the output **Raster** to be a function of the combined intensities of all three colors of the corresponding pixel in the input **Raster**, *(plus a constant that is equal to the rightmost value in the corresponding row of the image-processing matrix).*

Potential arithmetic overflow

It is unclear in the documentation what happens to the output color value if the value resulting from the matrix multiplication and the addition of the constant falls outside the range from 0 to 255. However, observation of the results indicates that rather than clipping the value to force it to be within the range from 0 to 255, the value is allowed to overflow and become corrupt. *(See Figure 6.)* Therefore, care must be exercised to avoid such overflow when setting the multiplicative values in the processing matrix.

Interesting visual effects. On the other hand, allowing the overflow to occur can lead to some interesting visual effects, as illustrated in Figure 10.

A variety of interesting effects

This processing approach can lead to a variety of interesting effects. One author says that this class can be used to create cubist-style images *(see Figure 10)*.

Limited knowledge of art. Given my limited knowledge of art, I will simply have to take the author's word on this.

Color inversion

As with some of the other image-filtering classes in the Java 2D API, the **BandCombineOp** class can easily be used to invert any or all of the colors in an image, producing an output such as that shown in the bottom panel of Figure 2.

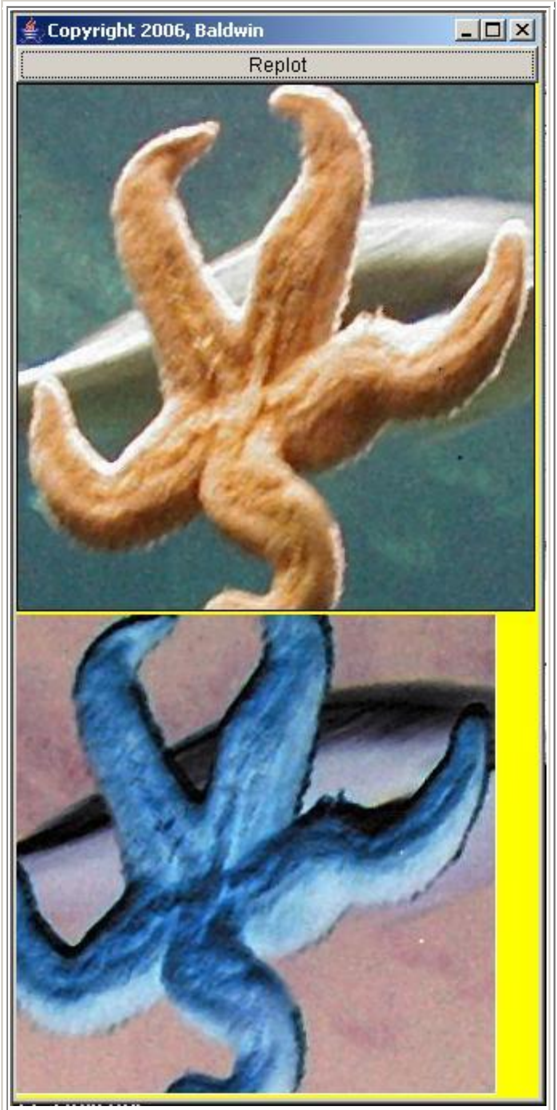
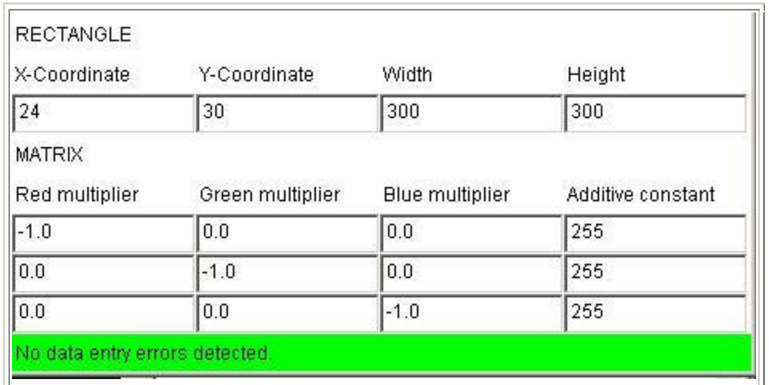


Figure 2

All three colors were inverted in Figure 2. In addition, Figure 2 illustrates the extraction and processing of a **Raster** object that is 300 pixels on each side. The raster was extracted from the original image such that the upper-left corner of the raster matches a pixel at a horizontal coordinate of 24 and a vertical coordinate of 30 in the original image.

The user input data

The user input data that was used to process this image is shown in Figure 3.



RECTANGLE

X-Coordinate	Y-Coordinate	Width	Height
24	30	300	300

MATRIX

Red multiplier	Green multiplier	Blue multiplier	Additive constant
-1.0	0.0	0.0	255
0.0	-1.0	0.0	255
0.0	0.0	-1.0	255

No data entry errors detected.

Figure 3

Figure 3 is a screen shot of the program GUI with the top portion cropped off to save space.

The RECTANGLE text fields

The values shown in the four RECTANGLE text fields in Figure 3 specify the location and size of the extracted **Raster** object described above.

The MATRIX text fields

The values in the twelve MATRIX text fields cause the red, green, and blue color values for each output pixel to be 255 minus the color value for the corresponding input pixel. You have learned in earlier lessons that this is the arithmetic process that can be used to invert the colors in an image.

Conversion to gray

Figure 4 shows the result of causing the red, green, and blue color values of each output pixel to be the average of all three color values of the corresponding input pixel. When all three color values for a pixel have the same value, the rendered color of the pixel is some shade of gray ranging from black to white.

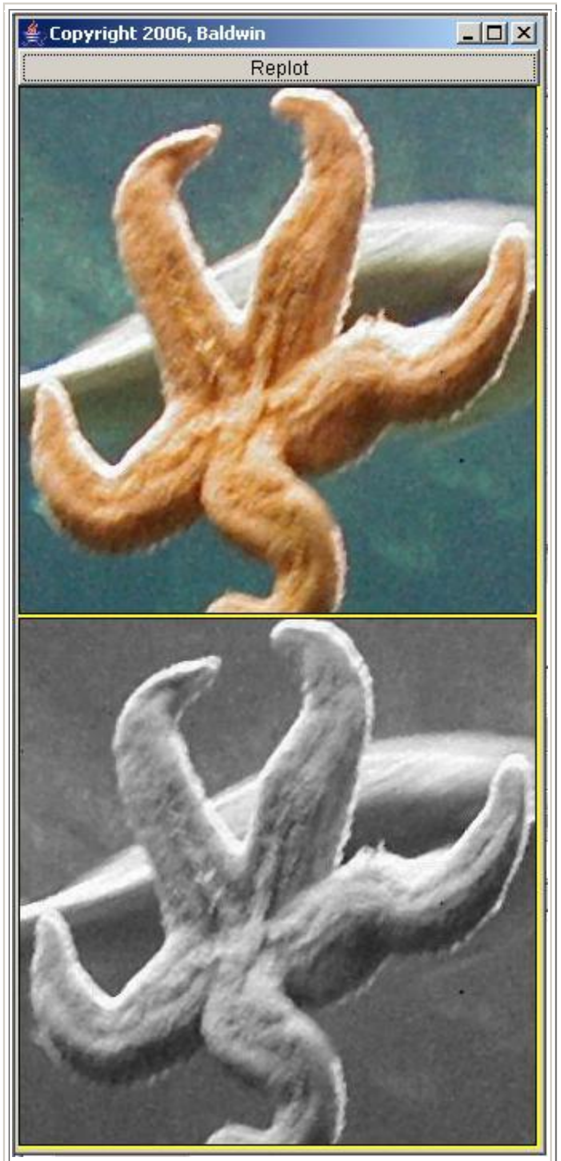
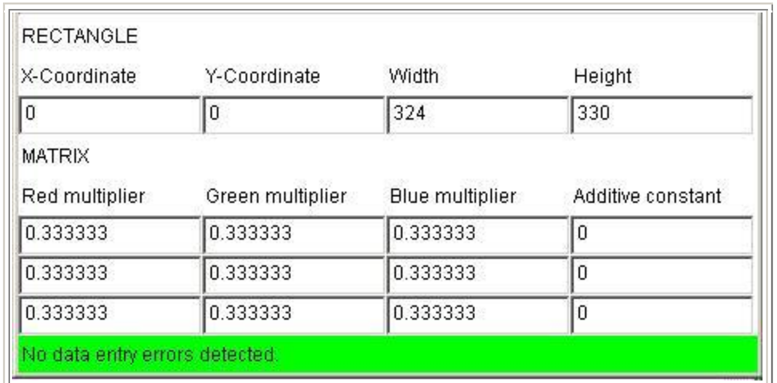


Figure 4

The user input data

The user input values used to produce the gray image in Figure 4 are shown in the screen shot of the GUI in Figure 5.


RECTANGLE

X-Coordinate	Y-Coordinate	Width	Height
0	0	324	330

MATRIX

Red multiplier	Green multiplier	Blue multiplier	Additive constant
0.333333	0.333333	0.333333	0
0.333333	0.333333	0.333333	0
0.333333	0.333333	0.333333	0

No data entry errors detected.

Figure 5

In this case, each of the three color values in each output pixel is the sum of one third of the red, green, and blue color values for the corresponding input pixel.

Arithmetic overflow

Figure 6 shows the same thing as Figure 4, except that each input color value in Figure 6 was multiplied by 0.5 *(instead of 0.33333)* before adding them together to produce the color values for the output pixel.

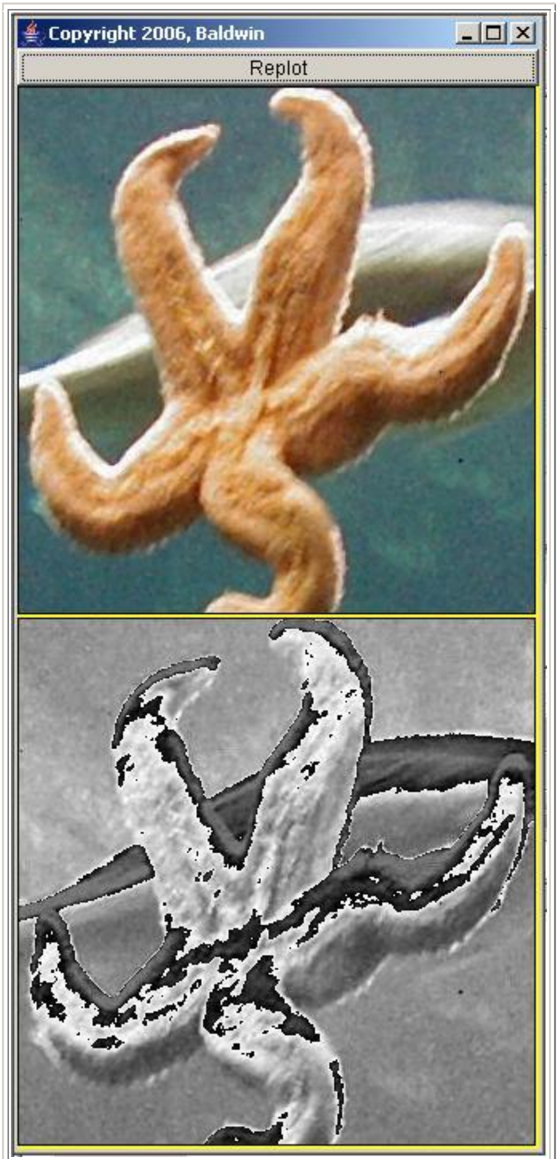
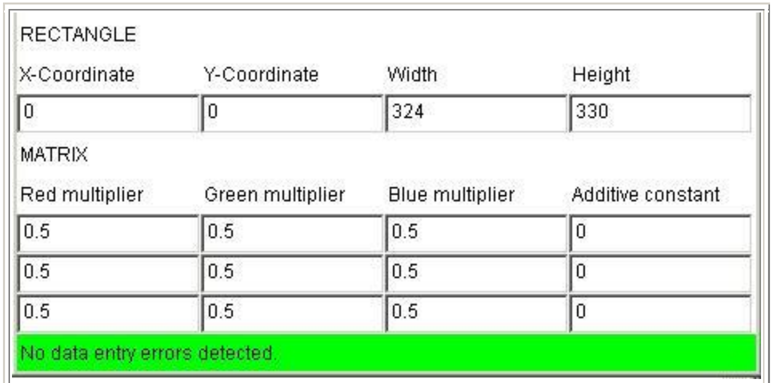


Figure 6

The user input data

The user input values that produced Figure 6 are shown in Figure 7.



RECTANGLE

X-Coordinate	Y-Coordinate	Width	Height
0	0	324	330

MATRIX

Red multiplier	Green multiplier	Blue multiplier	Additive constant
0.5	0.5	0.5	0
0.5	0.5	0.5	0
0.5	0.5	0.5	0

No data entry errors detected.

Figure 7

As you can see, this image-processing matrix caused the overall output image in Figure 6 to be somewhat brighter than the output image in Figure 4.

As you can also see, this resulted in arithmetic overflow for those output color values that exceeded a value of 255. As a result, light gray areas in Figure 4 became black or dark gray areas, often outlined with white, in Figure 6.

A gray negative

The image in Figure 4 is very similar to an old black and white photograph.

If you use an image-processing matrix that causes each output pixel color value to be the average of the corresponding input pixel color values *(as in Figure 4)*, and also invert the colors in the output, the resulting image is very similar to the *negative* film from an old black and white photograph. This is illustrated by the output image in Figure 8. Whereas Figure 4 is the *positive*, Figure 8 is the *negative*.

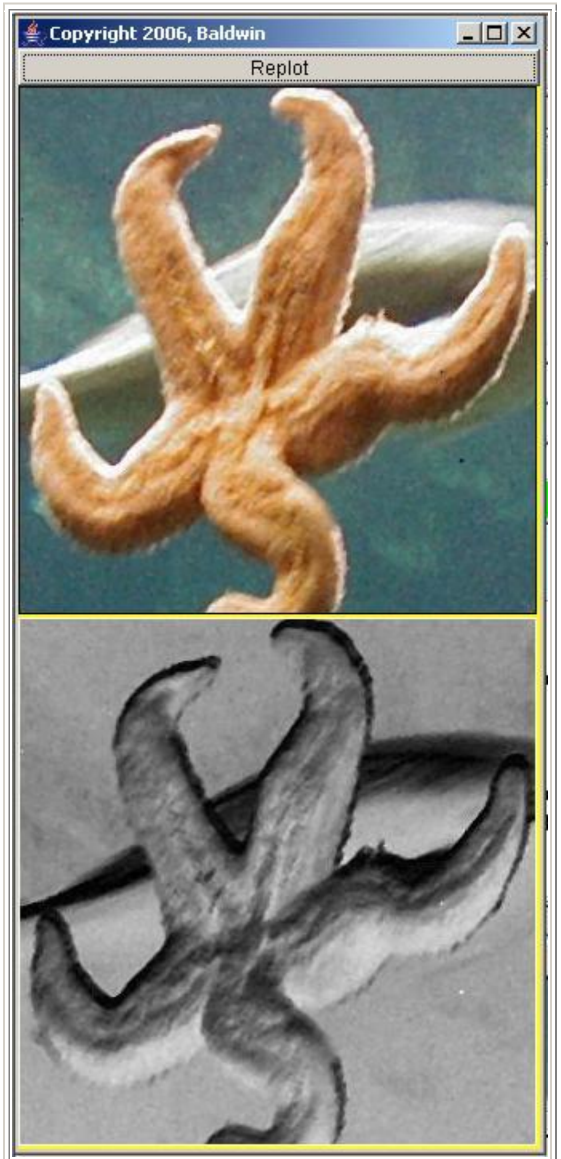
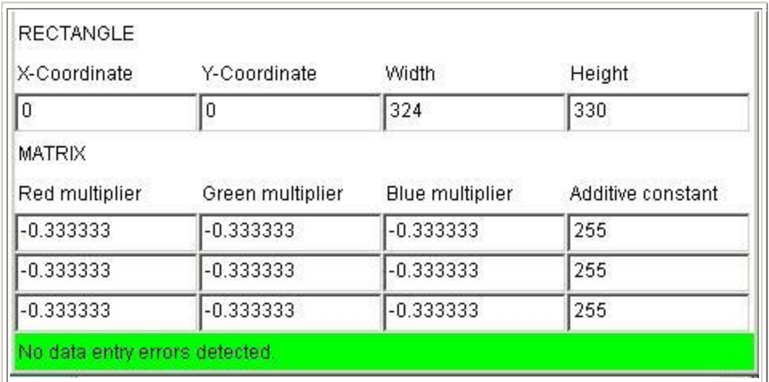


Figure 8

The user input data

Figure 9 shows the image-processing matrix that was used to produce the output image in Figure 8.



RECTANGLE

X-Coordinate	Y-Coordinate	Width	Height
0	0	324	330

MATRIX

Red multiplier	Green multiplier	Blue multiplier	Additive constant
-0.333333	-0.333333	-0.333333	255
-0.333333	-0.333333	-0.333333	255
-0.333333	-0.333333	-0.333333	255

No data entry errors detected.

Figure 9

If you compare Figure 8 with Figure 4, you should be able to easily see the difference between the *positive* version in Figure 4 and the *negative* version in Figure 8.

Now for some computer-generated artwork

For whatever it is worth, if I interpreted the previously-referenced article correctly, the output shown in Figure 10 is a *cubist-style* image. *(At least, it was produced using the same processing matrix as the one given in that article.)*

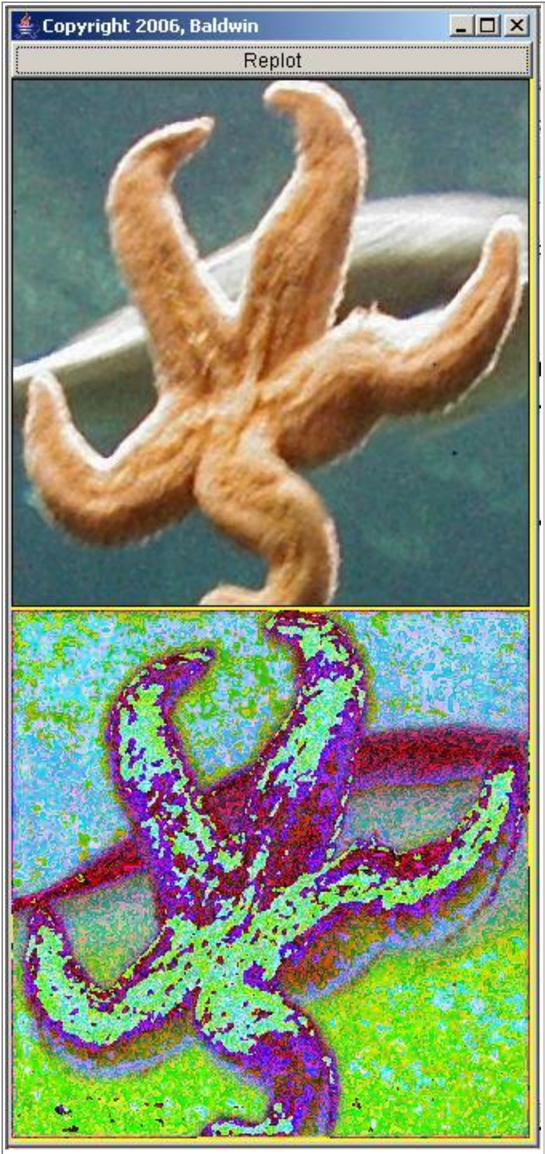
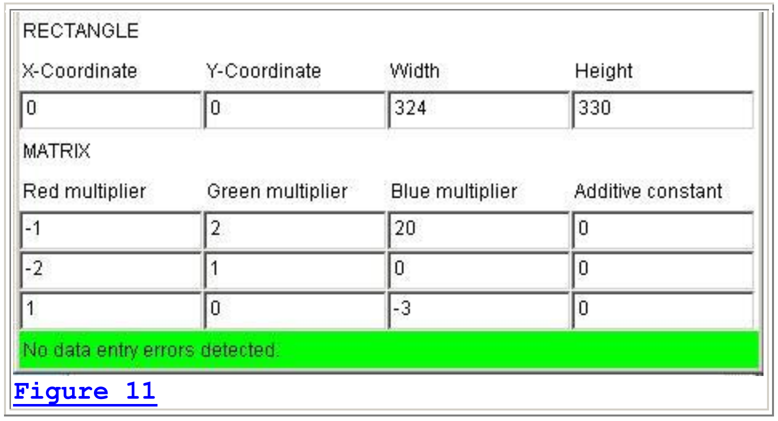


Figure 10

The image-processing matrix

The image-processing matrix that produced the output image in Figure 10 is shown in Figure 11.



RECTANGLE

X-Coordinate	Y-Coordinate	Width	Height
0	0	324	330

MATRIX

Red multiplier	Green multiplier	Blue multiplier	Additive constant
-1	2	20	0
-2	1	0	0
1	0	-3	0

No data entry errors detected.

Figure 11

However, as I mentioned earlier, given my limited knowledge of art, I will simply have to take the author's word that this is a *cubist-style* image. Whatever it is, it illustrates that it is possible to use the **BandCombineOp** class to produce some weird and interesting effects.

Now let's see some code.

Discussion and Sample Code

The program named ImgMod41

In this program, I will explain the behavior of a program named **ImgMod41.** A complete listing of the program is provided in Listing 14 near the end of the lesson.

Purpose

The purpose of this program is to illustrate the use of the **BandCombineOp** image-filtering class of the Java 2D API. *(See general comments in the class named **ImgMod038** that I explained in the earlier lesson entitled "Using the Java 2D LookupOp Filter Class to Process Images" (see References). Those comments apply to this program also.)*

Compatible with ImgMod05

The program named **ImgMod41** is compatible with the use of the framework program named **ImgMod05**. In other words, **ImgMod41** can be run under the control of **ImgMod05**.

The framework program named **ImgMod05** displays the original and the filtered images as shown in the top and bottom panels of Figure 2. **ImgMod05** also writes the filtered image into an output file in JPEG format. The name of the output file is **junk.jpg** and it is written into the current directory.

Running the program

Enter the following at the command line to run this program:

```
java ImgMod05 ImgMod41 ImageFileName
```

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

A user-input GUI

Image processing programs that run under the control of **ImgMod05** may provide a GUI for user data input. This makes it possible for the user to modify the behavior of the image-processing method each time the **Replot** button is clicked.

The GUI that is provided by this program is shown in Figure 1. This program GUI provides:

- User instructions.
- Text fields used to specify the upper-left corner, the width, and the height of a rectangle that is used to extract a **Raster**.
- Text fields used to specify the values in a 3x4 image-processing matrix having three rows and four columns.

Operates only on Raster objects

As explained earlier, the **BandCombineOp** filter can only operate on **Raster** objects. *(It cannot operate directly on **BufferedImage** objects.)* The rectangle described above is used to extract a rectangular **Raster** object from the original image.

Initialization

The rectangle values are initialized so that the rectangle is the same size as the image and overlays the entire image. In other words, the initial rectangle contains the complete image.

The matrix values described above are initialized so as to simply pass the input image through to the output without modification.

Image-filtering methodology

I explained the image-filtering methodology involving the multiplication of matrices earlier.

Program testing

This program was tested using J2SE 5.0 under WinXP.

Will explain in fragments

I will explain this program in fragments. A complete listing of the program is provided in Listing 14.

The class definition

The class definition begins in Listing 1.

```
class ImgMod41 extends Frame implements
ImgIntfc05{

  //Components used to construct the main
panel.
  // Components that require local access only
are defined
  // locally.  Others are defined here as
instance
  // variables.
  Panel mainPanel = new Panel();//main control
panel

  //Text fields for specifying the Rectangle
object values.
  TextField rectXcoorField = new
TextField("0");
  TextField rectYcoorField = new
TextField("0");
  TextField rectWidthField = new
TextField("0");
  TextField rectHeightField = new
TextField("0");

  //Text fields for specifying the matrix
values.
  //Top row
  TextField matrix00Field = new
TextField("1.0");
  TextField matrix01Field = new
TextField("0.0");
  TextField matrix02Field = new
TextField("0.0");
  TextField matrix03Field = new
TextField("0.0");

  //Middle row
  TextField matrix10Field = new
TextField("0.0");
  TextField matrix11Field = new
TextField("1.0");
  TextField matrix12Field = new
TextField("0.0");
  TextField matrix13Field = new
TextField("0.0");

  //Bottom row
  TextField matrix20Field = new
TextField("0.0");
  TextField matrix21Field = new
TextField("0.0");
```

```
  TextField matrix22Field = new
TextField("1.0");
  TextField matrix23Field = new
TextField("0.0");

  //The following Label is used to notify of
data entry
  // errors.
  String okMessage = "No data entry errors
detected.";
  Label errorMsg = new Label(okMessage);
```

Listing 1

The code in Listing 1 is straightforward and should not require further explanation beyond the embedded comments.

The constructor

The constructor is shown in its entirety in Listing 2.

```
  ImgMod41(){//constructor

    constructMainPanel();
    add(mainPanel);

    setTitle("Copyright 2006, R.G.Baldwin");
    setBounds(555,0,470,600);
    setVisible(true);

    //Define a WindowListener to terminate the
program.
    addWindowListener(
      new WindowAdapter(){
        public void windowClosing(WindowEvent
e){
          System.exit(1);
        }//end windowClosing
      }//end windowAdapter
    );//end addWindowListener
  }//end constructor
```

Listing 2

This is the primary constructor. It calls another method to actually construct the main panel so as to separate the construction of the GUI into easily understandable units.

The method named constructMainPanel

The method named **constructMainPanel** is shown in its entirety in Listing 3.   This method constructs the main GUI panel *(shown in Figure 1)* containing all of the controls. This method is called from the primary constructor in Listing 2.

```
  void constructMainPanel(){
    mainPanel.setLayout(new BorderLayout());

    //Create and add the instructional text to
the panel.
    // This text appears in a disabled text area
at the
    // top of the panel.
    String text ="COMBINING COLOR BAND DATA\n"
      + "This program illustrates the use of the
"
      + "BandCombineOp filter class of the Java
2D "
      + "API.\n\n"
      + "Specify the coordinates of the upper-
left corner "
      + "along with the width and the height of
a "
      + "rectangle that either exactly overlays
or fits "
      + "inside of the original image.  This
rectangle is "
      + "used to extract a rectangular Raster
with a "
      + "corresponding location and size from
the "
      + "image.\n\n"
      + "Specify the twelve values in a 3x4
processing "
      + "matrix and then click the Replot button
to "
      + "process the image.\n\n"
      + "The initial width and height values
match the "
      + "size of the image.  Set the width value
to 0 and "
      + "click the Replot button to recover the
width and "
      + "height of the image.\n\n"
      + "The red, green, and blue values from
each input "
      + "pixel plus a value of 1 is used to
construct a "
      + "1x4 column matrix that represents each
input "
      + "pixel.\n\n"
      + "Each output pixel is produced by
multiplying the "
      + "1x4 column matrix representing each
input pixel "
      + "by the 3x4 processing matrix.\n\n"
      + "Apparently output color values greater
than 255 "
      + "or less than 0 simply result in corrupt
values "
      + "in the output.";
    //Note:  The number of columns specified for
the
    // following TextArea is immaterial because
the
    // TextArea object is placed in the NORTH
location of
    // a BorderLayout.
    TextArea textArea = new TextArea(text,22,1,

TextArea.SCROLLBARS_NONE);
    mainPanel.add(textArea,BorderLayout.NORTH);
    textArea.setEnabled(false);

    //Construct the control panel and add it to
the Center
    // of the main panel.
    Panel controlPanel = new Panel();
    controlPanel.setLayout(new GridLayout(8,4));

    //Add a row of labels
    controlPanel.add(new Label("RECTANGLE"));
    controlPanel.add(new Label(""));
    controlPanel.add(new Label(""));
    controlPanel.add(new Label(""));

    //Add another row of labels
    controlPanel.add(new Label("X-Coordinate"));
    controlPanel.add(new Label("Y-Coordinate"));
    controlPanel.add(new Label("Width"));
    controlPanel.add(new Label("Height"));

    //Add the text fields for the rectangle
    controlPanel.add(rectXcoorField);
    controlPanel.add(rectYcoorField);
    controlPanel.add(rectWidthField);
    controlPanel.add(rectHeightField);

    //Add another row of labels
    controlPanel.add(new Label("MATRIX"));
    controlPanel.add(new Label(""));
    controlPanel.add(new Label(""));
    controlPanel.add(new Label(""));

    //Add another row of labels
    controlPanel.add(new Label("Red
multiplier"));
    controlPanel.add(new Label("Green
multiplier"));
    controlPanel.add(new Label("Blue
multiplier"));
    controlPanel.add(new Label("Additive
constant"));

    //Add top row of matix text fields
    controlPanel.add(matrix00Field);
    controlPanel.add(matrix01Field);
    controlPanel.add(matrix02Field);
    controlPanel.add(matrix03Field);

    //Add middle row of matrix text fields
    controlPanel.add(matrix10Field);
    controlPanel.add(matrix11Field);
    controlPanel.add(matrix12Field);
    controlPanel.add(matrix13Field);

    //Add bottom row of matrix text fields
    controlPanel.add(matrix20Field);
    controlPanel.add(matrix21Field);
    controlPanel.add(matrix22Field);
    controlPanel.add(matrix23Field);


mainPanel.add(controlPanel,BorderLayout.CENTER);

    //Add the errorMsg
    mainPanel.add(errorMsg,BorderLayout.SOUTH);
    errorMsg.setBackground(Color.GREEN);
  }//end constructMainPanel
```

Listing 3

Although somewhat long and tedious, the code in Listing 3 is straightforward. If you use Figure 1 as a guide, you should have no difficulty following the code in Listing 3.

The processImg method

The **processImg** method is shown in its entirety in Listing 4. This method must be defined to implement the **ImgIntfc05** interface, which is a requirement for compatibility with the framework program named **ImgMod05**.

The **processImg** method is called by the framework program named **ImgMod05**. Note that this method, in turn, calls the method named **processMainPanel**.

```
  public BufferedImage processImg(BufferedImage 
theImage){

    BufferedImage outputImage = 
processMainPanel(theImage);

    return outputImage;
  }//end processImg
```

Listing 4

The method named processMainPanel

This method is really the heart of the image-processing program named **ImgMod41**. The method begins in Listing 5.

```
  BufferedImage processMainPanel(BufferedImage
theImage){

    //Reset the error message to the default.
    errorMsg.setText(okMessage);
    errorMsg.setBackground(Color.GREEN);
```

Listing 5

This method uses the **BandCombineOp** image-filtering class to process the image according to the rectangle and matrix values provided by the user via the program GUI shown in Figure 1.

The method is called from within the method named **processImg** in Listing 4.

The code in Listing 5 initializes the error panel *(shown in green in Figure 1)* to its default condition.

Initialize the rectangle location and size

Listing 6 initializes the contents of the text fields that specify the rectangle so as to include the entire image within the rectangle.

```
    if((rectWidthField.getText().equals("0"))
               ||
(rectHeightField.getText().equals("0"))){
      rectWidthField.setText(""
+theImage.getWidth());
      rectHeightField.setText("" +
theImage.getHeight());
    }//end if
```

Listing 6

This initialization is performed only if either the width or height text fields contain a 0, which is the case at startup. These values can later be modified by the user.

> *(Also, for convenience, the user can later enter a 0 for the x-coordinate, the y-coordinate, and the width and then click the **Replot** to reset the rectangle values to the original image size.)*

Get the contents of the rectangle fields

Listing 7 gets the contents of the rectangle fields, performing some tests to confirm that the values are valid.

```
    int
rectXcoor,rectYcoor,rectWidth,rectHeight;
    try{
      rectXcoor =

Integer.parseInt(rectXcoorField.getText());
      rectYcoor =

Integer.parseInt(rectYcoorField.getText());
      rectWidth =

Integer.parseInt(rectWidthField.getText());
      rectHeight =

Integer.parseInt(rectHeightField.getText());
    }catch(java.lang.NumberFormatException e){
      //Bad data in the rectangle fields.
Process a 1x1
      // rectangle so that it will be obvious
to the user
      // that there is a problem.
      rectXcoor = rectYcoor = rectWidth =
rectHeight = 1;
      errorMsg.setText(
                      "Bad input data for the
rectangle.");
      errorMsg.setBackground(Color.RED);
    }//end catch

    int imageWidth = theImage.getWidth();
    int imageHeight = theImage.getHeight();

    //Code to confirm that the rectangle falls
inside the
    // image.
    if((rectXcoor < 0)||
       (rectYcoor < 0)||
       ((rectXcoor + rectWidth) > imageWidth)||
       ((rectYcoor + rectHeight) >
imageHeight))
    {
      //The rectangle falls outside the image.
Process a
      // 1x1 rectangle so that it will be
obvious to the
      // user that there is a problem.
      rectXcoor = rectYcoor = rectWidth =
rectHeight = 1;
      errorMsg.setText(
                 "The rectangle falls outside
the image.");
      errorMsg.setBackground(Color.RED);
    }//end if
```

Listing 7

If the data entered into the rectangle text fields cannot be parsed to produce values of type **int**, or the location and size of the rectangle is outside the bounds of the image, the green error panel at the bottom of Figure 1 turns red and displays an error message. To correct the problem, simply enter valid data and click the **Replot** button again.

Get the image-processing matrix values

Listing 8 gets the values for the image-processing matrix from the twelve text fields at the bottom of Figure 1.

```
    float
matrix00,matrix01,matrix02,matrix03,matrix10,

matrix11,matrix12,matrix13,matrix20,matrix21,
          matrix22,matrix23;
    try{
      matrix00 =
Float.parseFloat(matrix00Field.getText());
      matrix01 =
Float.parseFloat(matrix01Field.getText());
      matrix02 =
Float.parseFloat(matrix02Field.getText());
      matrix03 =
Float.parseFloat(matrix03Field.getText());
      matrix10 =
Float.parseFloat(matrix10Field.getText());
      matrix11 =
Float.parseFloat(matrix11Field.getText());
      matrix12 =
Float.parseFloat(matrix12Field.getText());
      matrix13 =
Float.parseFloat(matrix13Field.getText());
      matrix20 =
Float.parseFloat(matrix20Field.getText());
      matrix21 =
Float.parseFloat(matrix21Field.getText());
      matrix22 =
Float.parseFloat(matrix22Field.getText());
      matrix23 =
Float.parseFloat(matrix23Field.getText());
    }catch(java.lang.NumberFormatException e){
      //Bad input data for the matrix.  Cause
the output
      // image to be black so that it will be
obvious to
      // the user that there is a problem.
      matrix00 = matrix01 = matrix02 = matrix03
=
      matrix10 = matrix11 = matrix12 = matrix13
=
      matrix20 = matrix21 = matrix22 = matrix23
= 0.0f;
      errorMsg.setText("Bad input data for the
matrix.");
```

```
      errorMsg.setBackground(Color.RED);
    }//end catch

    //Now populate the matrix
    float[][] matrix =

{{matrix00,matrix01,matrix02,matrix03},

{matrix10,matrix11,matrix12,matrix13},

{matrix20,matrix21,matrix22,matrix23}
                    };
```

Listing 8

If the values entered by the user into the text fields can't be parsed to produce valid values of type **float**, the error panel in the bottom of Figure 1 turns red and displays an error message. To correct the problem, simply correct the entry in the text field and click the **Replot** button.

Set up the matrix array

Once the values have been fetched from the text fields to produce values of type **float**, the code in Listing 8 uses those values to create and populate a 2D array object, which will be used later in Listing 10 to create the image-filtering object.

Get the image data in a Raster object

Listing 9 invokes the method named **getData** on the **BufferedImage** object containing the image to extract and save a rectangular set of pixel data that matches the location and size of the rectangle.

```
    Raster inputRaster = theImage.getData(new
Rectangle(

rectXcoor,rectYcoor,rectWidth,rectHeight));
```

Listing 9

According to Sun, the **getData** method *"Returns the image as one large tile."* This **Raster** object will be used later in Listing 11 to create a destination raster for the filtered image.

Create the image-filtering object

Listing 10 instantiates a new image-filtering object of type **BandCombineOp**.

```
    BandCombineOp filterObj =
                            new
BandCombineOp(matrix,null);
```

Listing 10

The first parameter to the constructor for the **BandCombineOp** class is the image-processing matrix created in Listing 8.

The second parameter allows for the use of rendering hints, but that capability is not used in this program.

Create a destination Raster object

Listing 11 invokes the **createCompatibleDestRaster** method on the filter object to create a zeroed destination **Raster** with the correct size and number of color bands.

```
    WritableRaster destinationRaster =

filterObj.createCompatibleDestRaster(inputRaster);
```

Listing 11

The createCompatibleDestRaster method

This method returns a reference to an object of type **WritableRaster**, which is a subclass of **Raster**. Part of what Sun has to say about the **WritableRaster** class is shown in Figure 12.

This class extends **Raster** to provide pixel writing capabilities. Refer to the class comment for **Raster** for descriptions of how a **Raster** stores pixels.

The constructors of this class are protected. To instantiate a **WritableRaster**, use one of the **createWritableRaster** factory methods in the **Raster** class.

Figure 12

Obviously, it is also possible to create a **WritableRaster** object by invoking the **createCompatibleDestRaster** method of the **BandCombineOp** class as was done in Listing 11.

Apply the filter to the image

Listing 12 applies the filter to the raster that contains the image and deposits the filtered image in the destination raster.

```

filterObj.filter(inputRaster,destinationRaster);
```

Listing 12

Return the filtered image

Listing 13 converts the destination raster to a **BufferedImage** object and returns a reference to the **BufferedImage** object.

```
    return new
BufferedImage(theImage.getColorModel(),
                             destinationRaster,
                             false,
                             null);

  }//end processMainPanel

}//end class ImgMod41
```

Listing 13

The first parameter causes the **ColorModel** for the output image to be the same as the **ColorModel** for the input image. The second parameter is a reference to the **WritableRaster** object that contains the filtered image.

The third parameter indicates that the color values have not been pre-multiplied by the alpha values. The fourth parameter allows for the inclusion of some properties in a **Hashtable** object. *(I will leave it as an exercise for the reader to investigate and understand the meaning and purpose of the third and fourth parameters.)*

End of the method and end of the program

Listing 13 also signals the end of the **processMainPanel** method and the end of the program named **ImgMod41**.

Run the Program

I encourage you to copy the code from Listing 14 into your text editor. Compile the code and execute it. Experiment with it, making changes, and observing the results of your changes.

Remember, you will also need to compile the code for the framework program named **ImgMod05** and the interface named **ImgIntfc05**. You will find that source code in the earlier lesson entitled "A Framework for Experimenting with Java 2D Image-Processing Filters" *(see References)*.

You will also need one or more JPEG, GIF, PNG, or BMP image files to experiment with. You should have no difficulty finding such files at a variety of locations on the web. I recommend that you stick with relatively small images so that both the original image and the processed image will fit in the vertical space on your screen in the format shown in Figure 2.

Summary

In this lesson, I provided and explained an image-processing program named **ImgMod41** that is compatible with the framework program named **ImgMod05**.

The purpose of this program is to show you how to write such programs, and also to illustrate a variety of different uses for the **BandCombineOp** class of the Java 2D API.

I also showed you how to extract a **Raster** object from a **BufferedImage**, how to filter the raster, and how to convert the filtered result back into a **BufferedImage.**

What's Next?

Future lessons in this series will teach you how to use the following image-filtering classes from the Java 2D API:

- **ConvolveOp**
- **RescaleOp**
- **ColorConvertOp**

References

- 400 Processing Image Pixels using Java, Getting Started
- 402 Processing Image Pixels using Java, Creating a Spotlight
- 404 Processing Image Pixels Using Java: Controlling Contrast and Brightness
- 406 Processing Image Pixels, Color Intensity, Color Filtering, and Color Inversion
- 408 Processing Image Pixels, Performing Convolution on Images
- 410 Processing Image Pixels, Understanding Image Convolution in Java
- 412 Processing Image Pixels, Applying Image Convolution in Java, Part 1
- 414 Processing Image Pixels, Applying Image Convolution in Java, Part 2
- 416 Processing Image Pixels, An Improved Image-Processing Framework in Java
- 450 A Framework for Experimenting with Java 2D Image-Processing Filters
- 452 Using the Java 2D LookupOp Filter Class to Process Images
- 454 Using the Java 2D AffineTransformOp Filter Class to Process Images
- 456 Using the Java 2D LookupOp Filter Class to Scramble and Unscramble Images

Complete Program Listing

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

```
/*File ImgMod41.java
Copyright 2006, R.G.Baldwin

The purpose of this class is to illustrate the use of the
BandCombineOp image-filtering class of the Java 2D API.

See general comments in the class named ImgMod038.
```

```
This class is compatible with the use of the framework
program named ImgMod05.

The framework program named ImgMod05 displays the original
and the modified images.  It also writes the modified image
into an output file in JPEG format.  The name of the output
file is junk.jpg and it is written into the current
directory.

Image processing programs such as this one may provide a
GUI for data input making it possible for the user to
modify the behavior of the image processing method each
time the Replot button is clicked.  Such a GUI is provided
for this program.

Enter the following at the command line to run this
program:

java ImgMod05 ImgMod41 ImageFileName

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

This program creates a GUI containing:

User instructions
Text fields used to specify the upper-left corner, the
 width, and the height of a rectangle.
Text fields used to specify the values in a 3x4 processing
 matrix having three rows and four columns.

Unlike some of the other image-filtering classes in the
Java 2D API that can operate either directly on
BufferedImage objects or on Raster objects, the
BandCombineOp filter can only operate on Raster objects.

The rectangle is used to extract a rectangular Raster
object from the original image.

The rectangle values are initialized so that the rectangle
is the same size as the image and overlays the entire
image.  In other words, the rectangle contains the complete
image.

The matrix values are initialized so as to simply pass the
input image through to the output without modification..

The red, green, and blue values of each pixel are treated
as a column matrix.  A 1 is appended onto the end of the
column matrix producing a set of four-element column
matrices that represents each pixel in the input Raster
object.

Each pixel in the output Raster is produced by multiplying
the 3x4 processing matrix by the 4x1 column matrix that
represents the corresponding pixel in the input Raster.
```

```
This makes it possible to cause the intensity of each color
in each pixel of the output Raster to be a function of the
intensities of all three colors of the corresponding pixel
in the input Raster, plus a constant that is equal to the
rightmost value in the corresponding row of the processing
matrix.

It is unclear in the documentation what happens to the
output color value if the value resulting from the matrix
multiplication falls outside the range from 0 to 255.
However, observation of the results suggests that rather
than clipping the value to be within the range from 0 to
255, the value is allowed to become corrupt.Therefore, care
must be exercised to avoid such overflow when setting the
multiplicative values in the processing matrix.

Tested using J2SE 5.0 under WinXP.
**********************************************************/

import java.awt.image.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class ImgMod41 extends Frame implements ImgIntfc05{

  //Components used to construct the main panel.
  // Components that require local access only are defined
  // locally.  Others are defined here as instance
  // variables.
  Panel mainPanel = new Panel();//main control panel

  //Text fields for specifying the Rectangle object values.
  TextField rectXcoorField = new TextField("0");
  TextField rectYcoorField = new TextField("0");
  TextField rectWidthField = new TextField("0");
  TextField rectHeightField = new TextField("0");

  //Text fields for specifying the matrix values.
  //Top row
  TextField matrix00Field = new TextField("1.0");
  TextField matrix01Field = new TextField("0.0");
  TextField matrix02Field = new TextField("0.0");
  TextField matrix03Field = new TextField("0.0");

  //Middle row
  TextField matrix10Field = new TextField("0.0");
  TextField matrix11Field = new TextField("1.0");
  TextField matrix12Field = new TextField("0.0");
  TextField matrix13Field = new TextField("0.0");

  //Bottom row
  TextField matrix20Field = new TextField("0.0");
  TextField matrix21Field = new TextField("0.0");
  TextField matrix22Field = new TextField("1.0");
  TextField matrix23Field = new TextField("0.0");

  //The following Label is used to notify of data entry
  // errors.
  String okMessage = "No data entry errors detected.";
  Label errorMsg = new Label(okMessage);

  //-----------------------------------------------------//

  //This is the primary constructor.  It calls another
  // method to construct the main panel so as to separate
  // the construction of the GUI into easily
  // understandable units.
  ImgMod41(){//constructor

    constructMainPanel();
    add(mainPanel);

    setTitle("Copyright 2006, R.G.Baldwin");
    setBounds(555,0,470,600);
    setVisible(true);

    //Define a WindowListener to terminate the program.
    addWindowListener(
      new WindowAdapter(){
        public void windowClosing(WindowEvent e){
          System.exit(1);
        }//end windowClosing
      }//end windowAdapter
    );//end addWindowListener
  }//end constructor
  //-----------------------------------------------------//

  //This method constructs the main panel containing all of
  // the controls.  This method is called from the primary
  // constructor.
  void constructMainPanel(){
    mainPanel.setLayout(new BorderLayout());

    //Create and add the instructional text to the panel.
    // This text appears in a disabled text area at the
    // top of the panel.
    String text ="COMBINING COLOR BAND DATA\n"
      + "This program illustrates the use of the "
      + "BandCombineOp filter class of the Java 2D "
      + "API.\n\n"
      + "Specify the coordinates of the upper-left corner "
      + "along with the width and the height of a "
      + "rectangle that either exactly overlays or fits "
      + "inside of the original image.  This rectangle is "
      + "used to extract a rectangular Raster with a "
      + "corresponding location and size from the "
      + "image.\n\n"
      + "Specify the twelve values in a 3x4 processing "
      + "matrix and then click the Replot button to "
      + "process the image.\n\n"
      + "The initial width and height values match the "
      + "size of the image.  Set the width value to 0 and "
      + "click the Replot button to recover the width and "
      + "height of the image.\n\n"
      + "The red, green, and blue values from each input "
      + "pixel plus a value of 1 is used to construct a "
      + "1x4 column matrix that represents each input "
      + "pixel.\n\n"
      + "Each output pixel is produced by multiplying the "
      + "1x4 column matrix representing each input pixel "
      + "by the 3x4 processing matrix.\n\n"
      + "Apparently output color values greater than 255 "
      + "or less than 0 simply result in corrupt values "
      + "in the output.";

    //Note:  The number of columns specified for the
    // following TextArea is immaterial because the
    // TextArea object is placed in the NORTH location of
    // a BorderLayout.
    TextArea textArea = new TextArea(text,22,1,
                             TextArea.SCROLLBARS_NONE);
    mainPanel.add(textArea,BorderLayout.NORTH);
    textArea.setEnabled(false);

    //Construct the control panel and add it to the Center
    // of the main panel.
    Panel controlPanel = new Panel();
    controlPanel.setLayout(new GridLayout(8,4));

    //Add a row of labels
    controlPanel.add(new Label("RECTANGLE"));
    controlPanel.add(new Label(""));
    controlPanel.add(new Label(""));
    controlPanel.add(new Label(""));

    //Add another row of labels
    controlPanel.add(new Label("X-Coordinate"));
    controlPanel.add(new Label("Y-Coordinate"));
    controlPanel.add(new Label("Width"));
    controlPanel.add(new Label("Height"));

    //Add the text fields for the rectangle
    controlPanel.add(rectXcoorField);
    controlPanel.add(rectYcoorField);
    controlPanel.add(rectWidthField);
    controlPanel.add(rectHeightField);

    //Add another row of labels
    controlPanel.add(new Label("MATRIX"));
    controlPanel.add(new Label(""));
    controlPanel.add(new Label(""));
    controlPanel.add(new Label(""));

    //Add another row of labels
    controlPanel.add(new Label("Red multiplier"));
    controlPanel.add(new Label("Green multiplier"));
    controlPanel.add(new Label("Blue multiplier"));
    controlPanel.add(new Label("Additive constant"));

    //Add top row of matix text fields
    controlPanel.add(matrix00Field);
    controlPanel.add(matrix01Field);
    controlPanel.add(matrix02Field);
    controlPanel.add(matrix03Field);

    //Add middle row of matrix text fields
    controlPanel.add(matrix10Field);
    controlPanel.add(matrix11Field);
    controlPanel.add(matrix12Field);
    controlPanel.add(matrix13Field);

    //Add bottom row of matrix text fields
    controlPanel.add(matrix20Field);
    controlPanel.add(matrix21Field);
    controlPanel.add(matrix22Field);
    controlPanel.add(matrix23Field);

    mainPanel.add(controlPanel,BorderLayout.CENTER);

    //Add the errorMsg
    mainPanel.add(errorMsg,BorderLayout.SOUTH);
    errorMsg.setBackground(Color.GREEN);
  }//end constructMainPanel
  //-----------------------------------------------------//

  //This method processes the image according to the
  // rectangle and matrix values provided by the user.
  //This method uses the BandCombineOp image-filtering
  // class to process the image.  The method is called from
  // within the method named processImg, which is the
  // primary image processing method in this program.  The
  // method named processImg is called by the framework
  // program named ImgMod05.
  BufferedImage processMainPanel(BufferedImage theImage){

    //Reset the error message to the default.
    errorMsg.setText(okMessage);
    errorMsg.setBackground(Color.GREEN);

    //Initialize the contents of the text fields that
    // specify the rectangle so as to include the entire
    // image within the rectangle.  This initialization is
    // performed only if either the width or height text
    // fields contain a 0, which is the case at startup.
    // These values can later be modified by the user.
    // Also, the user can enter a 0 for the width and
    // click Replot to get back to the original image size.
    if((rectWidthField.getText().equals("0"))
               || (rectHeightField.getText().equals("0"))){
      rectWidthField.setText("" +theImage.getWidth());
      rectHeightField.setText("" + theImage.getHeight());
    }//end if
    //Get the contents of the rectangle fields.
    int rectXcoor,rectYcoor,rectWidth,rectHeight;
    try{
      rectXcoor =
                Integer.parseInt(rectXcoorField.getText());
      rectYcoor =
                Integer.parseInt(rectYcoorField.getText());
      rectWidth =
                Integer.parseInt(rectWidthField.getText());
      rectHeight =
               Integer.parseInt(rectHeightField.getText());
    }catch(java.lang.NumberFormatException e){
      //Bad data in the rectangle fields.  Process a 1x1
      // rectangle so that it will be obvious to the user
      // that there is a problem.
      rectXcoor = rectYcoor = rectWidth = rectHeight = 1;
      errorMsg.setText(
                      "Bad input data for the rectangle.");
      errorMsg.setBackground(Color.RED);
    }//end catch

    int imageWidth = theImage.getWidth();
    int imageHeight = theImage.getHeight();

    //Code to confirm that the rectangle falls inside the
    // image.
    if((rectXcoor < 0)||
       (rectYcoor < 0)||
       ((rectXcoor + rectWidth) > imageWidth)||
       ((rectYcoor + rectHeight) > imageHeight))
    {
      //The rectangle falls outside the image.  Process a
      // 1x1 rectangle so that it will be obvious to the
      // user that there is a problem.
      rectXcoor = rectYcoor = rectWidth = rectHeight = 1;
      errorMsg.setText(
                 "The rectangle falls outside the image.");
      errorMsg.setBackground(Color.RED);
    }//end if

    //Get the data from the text fields for the matrix.
    float matrix00,matrix01,matrix02,matrix03,matrix10,
          matrix11,matrix12,matrix13,matrix20,matrix21,
          matrix22,matrix23;
    try{
      matrix00 = Float.parseFloat(matrix00Field.getText());
      matrix01 = Float.parseFloat(matrix01Field.getText());
      matrix02 = Float.parseFloat(matrix02Field.getText());
      matrix03 = Float.parseFloat(matrix03Field.getText());
      matrix10 = Float.parseFloat(matrix10Field.getText());
      matrix11 = Float.parseFloat(matrix11Field.getText());
      matrix12 = Float.parseFloat(matrix12Field.getText());
      matrix13 = Float.parseFloat(matrix13Field.getText());
      matrix20 = Float.parseFloat(matrix20Field.getText());
      matrix21 = Float.parseFloat(matrix21Field.getText());
      matrix22 = Float.parseFloat(matrix22Field.getText());
      matrix23 = Float.parseFloat(matrix23Field.getText());
    }catch(java.lang.NumberFormatException e){
      //Bad input data for the matrix.  Cause the output
      // image to be black so that it will be obvious to
      // the user that there is a problem.
      matrix00 = matrix01 = matrix02 = matrix03 =
      matrix10 = matrix11 = matrix12 = matrix13 =
      matrix20 = matrix21 = matrix22 = matrix23 = 0.0f;
      errorMsg.setText("Bad input data for the matrix.");
      errorMsg.setBackground(Color.RED);
    }//end catch

    //Now populate the matrix
    float[][] matrix =
                    {{matrix00,matrix01,matrix02,matrix03},
                     {matrix10,matrix11,matrix12,matrix13},
                     {matrix20,matrix21,matrix22,matrix23}
                    };

    //Note:  Unlike some of the other filters in the Java
    // 2D API that can operate either directly on
    // BufferedImage objects or on Raster objects, the
    // BandCombineOp filter can only operate on Raster
    // objects.

    //Get the Raster object that contains the image data
    // inside the specified Rectangle object.
    Raster inputRaster = theImage.getData(new Rectangle(
                rectXcoor,rectYcoor,rectWidth,rectHeight));

    //Create the filter object.  The second parameter
    // allows for specification of rendering hints.
    BandCombineOp filterObj =
                            new BandCombineOp(matrix,null);

    //Create a zeroed destination Raster with the correct
    // size and number of bands.
    WritableRaster destinationRaster =
         filterObj.createCompatibleDestRaster(inputRaster);

    //Apply the filter
    filterObj.filter(inputRaster,destinationRaster);

    //Convert the destination raster to a BufferedImage and
    // return it.  The first parameter causes the
    // ColorModel for the output image to be the same as
    // the ColorModel for the input image.  The third
    // parameter indicates that the color values have not
    // been premultiplied by the alpha values.  The fourth
    // parameter allows for the inclusion of some
    // properties in a Hashtable object.
    return new BufferedImage(theImage.getColorModel(),
                             destinationRaster,
                             false,
                             null);
```

```
  }//end processMainPanel
  //-----------------------------------------------------//

  //The following method must be defined to implement the
  // ImgIntfc05 interface.  It is called by the framework
  // program named ImgMod05.
  public BufferedImage processImg(BufferedImage theImage){

    BufferedImage outputImage = processMainPanel(theImage);

    return outputImage;
  }//end processImg
}//end class ImgMod41
```

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 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 2D image pixel framework filter BandCombineOp

-end-