Processing Image Pixels, Performing Convolution on Images

Learn to write Java programs that use convolution (flat filters and Gaussian filters) to smooth or blur an image. Also learn how to write jpg files containing specialized images that are useful for testing image-processing programs.

Published: July 26, 2005
By Richard G. Baldwin

Java Programming, Notes # 408

Preface

Next in a series

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

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

Not a lesson on JAI

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

> *(However, this lesson does present two programs that make heavy use of the JAI API without providing much in the way of an explanation as to how they do what*

they do. These two programs are used to create jpg files, which in turn are used as input to the two primary image-processing programs that I will explain in this lesson.)

You will need a driver program

The lesson entitled Processing Image Pixels Using Java: Controlling Contrast and Brightness provided and explained a program named **ImgMod02a** that makes it easy to:

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

ImgMod02a serves as a driver that controls the execution of a second program that actually processes the pixels.

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

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.

Display format

The output shown in Figure 1 was produced by the driver program named **ImgMod02a** and the image-processing program named **ImgMod24**.

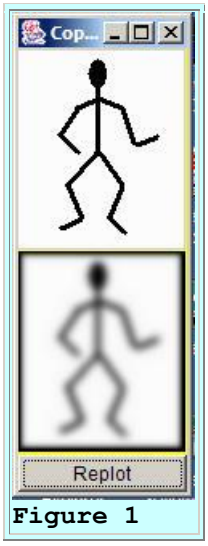


Figure 1

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

An interactive image-processing program

The image-processing program illustrated by Figure 1 allows the user to interactively control certain aspects of the process that I will describe later.

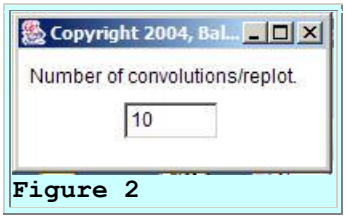


Figure 2

Figure 2 shows the control panel through which the user interactively controls that process. The user enters an integer value into the text field in Figure 2 and then presses the **Replot** button at the bottom of Figure 1 to cause the process to be rerun using the new input value.

Theoretical basis and practical implementation

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

Background Information

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

The earlier lesson introduced and explained the concept of a pixel. In addition, the lesson provided a brief discussion of image files, and indicated that the program named **ImgMod02a** is compatible with ***gif*** files, ***jpg*** files, and possibly some other file formats as well.

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

A three-dimensional array of pixel data as type int

The driver program named **ImgMod02a**:

- Extracts the pixels from an image file.
- Converts the pixel data to type **int.**

- Stores the pixel data in a three-dimensional array of type **int** that is well suited for processing.
- Passes the three-dimensional array object's reference to an image-processing program.
- Receives a reference to a three-dimensional array object containing processed pixel data from the image-processing program.
- Displays the original image and the processed image in a stacked display as shown in Figure 1.
- 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-processed image along with the original image.

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

Will concentrate on the three-dimensional array of type int

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

A grid of colored pixels

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

Fundamentals

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

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

Convolution in one dimension

The earlier lesson entitled Convolution and Frequency Filtering in Java taught you about performing convolution in one dimension. In that lesson, I showed you how to apply a

convolution operator to a sampled time series in one dimension. As you may recall, the mathematical process in one dimension involves the following steps:

- Register the n-point convolution operator with the first **n** samples in the time series.
- Compute an output point value, which is the sum of the products of the convolution operator values and the corresponding time series values.
- Move the convolution operator one step forward, registering it with the next **n** samples in the time series and compute the next output point value as a sum of products.
- Repeat this process until all samples in the time series have been processed.

Convolution in two dimensions

Convolution in two dimensions involves essentially the same steps except that in this case we are dealing with three different 3D sampled surfaces and a 3D convolution operator instead of a simple sampled time series.

> *(There is a red surface, a green surface, and a blue surface, each of which must be processed. Each surface has width and height corresponding to the first two dimensions of the 3D surface. In addition, each sampled value that represents the surface can be different. This constitutes the third dimension of the surface. There is also an alpha or transparency surface that could be processed, but the programs in this lesson don't process the alpha surface. Similarly, the convolution operator has three dimensions corresponding to width, height, and the values of the coefficients in the operator.)*

Lots of arithmetic required

Because each surface has three dimensions and there are three surfaces to be processed by a 3D convolution operator, the amount of arithmetic that must be performed can be quite large. Therefore, we will be looking for ways to make the arithmetic process more efficient than might be the case if we were to approach the problem simply using a brute-force multiply-add approach.

Steps in the processing

Basically, the steps involved in processing each of the three surfaces to produce an output surface consist of:

- Register the 2D aspect *(width and height)* of the convolution operator with the first 2D area centered on the first row of samples on the input surface.
- Compute a point for the output surface, by computing the sum of the products of the convolution operator values and the corresponding input surface values.
- Move the convolution operator one step forward along the row, registering it with the next 2D area on the surface and compute the next point on the output surface as a sum of products. When that row has been completely processed, move the convolution operator

to the beginning of the next row, registering with the corresponding 2D area on the input surface and compute the next point for the output surface.
- Repeat this process until all samples in the surface have been processed.

Repeat once for each color surface

Repeat the above set of steps three times, once for each of the three color surfaces.

Watch out for the edges

As you will see later, special care must be taken to avoid having the edges of the convolution operator extend outside the boundaries of the input surface.

The size of the convolution operator

One of the most important issues in performing convolution on images has to do with the ability to vary the 2D aspect of the size of the convolution operator. The two programs that I will explain in this lesson approach this process in two different ways.

A Gaussian shape with a round footprint

The program named **ImgMod24** begins with a flat 3x3 square convolution operator and allows the user to effectively increase the size and the 3D shape of the operator by performing multiple successive convolution operations on the input surface. In this case, the effective 3D shape of the convolution operator approaches a Gaussian shape with a round footprint as more and more successive convolutions are performed.

(I will explain what I mean by a Gaussian shape with a round footprint later.)

A totally flat convolution operator

The program named **ImgMod12** allows the user to specify the size of a flat rectangular convolution operator. For small operator sizes, the shape of the operator can be a rectangle. For large operator sizes, the shape of the operator is constrained to be a perfect square. In all cases, the 3D shape of the operator remains flat. This greatly reduces the arithmetic required to perform the processing.

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

Five programs and one interface

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

I will present and explain two new Java programs named **ImgMod12** and **ImgMod24** in this lesson. These programs, when run under control of the program named **ImgMod02a**, will produce outputs similar to Figure 1.

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

In addition, I will present, but will not fully explain, two programs named **ImgMaker01** and **ImgMaker02**. These two programs will be used to produce output jpg files that are useful in illustrating and explaining the behavior of the two image-processing programs.

The processImg method

The programs named **ImgMod12** and **ImgMod24**, *(and all image-processing programs that are capable of being driven by **ImgMod02a**)*, must implement the interface named **ImgIntfc02**. That interface declares a single method named **processImg**, which must be defined by all implementing classes.

When the user runs the program named **ImgMod02a**, that program instantiates an object of the image-processing program class and invokes the **processImg** method on that object.

A three-dimensional array containing the pixel data for the image is passed to the **processImg** method. The **processImg** method returns a three-dimensional array containing the pixel data for a processed version of the original image.

A *before* and *after* display

When the **processImg** method returns, the driver program named **ImgMod02a** causes the original image and the processed image to be displayed in a frame with the original image above the processed image as shown earlier in Figure 1.

Usage information for ImgMod12 and ImgMod24

To use the program named **ImgMod02a** to drive the program named **ImgMod12**, enter the following at the command line:

```
java ImgMod02a ImgMod12 ImagePathAndFileName
```

To use the program named **ImgMod02a** to drive the program named **ImgMod24**, enter the following at the command line:

```
java ImgMod02a ImgMod24 ImagePathAndFileName
```

The image file

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

> *(You should be able to right-click on the image in Figure 16 to download and save the image locally. Then you should be able to replicate the output produced in Figure 1 by running the program named **ImgMod24** and specifying 10 convolutions to process that image.)*

Image display format

When the program is started, the original image and the processed image for the default processing parameters are displayed in a frame with the original image above the processed image as shown in Figure 1.

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

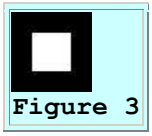
The image-processing programs named **ImgMod12** and **ImgMod24** provide a GUI for user input. A sample of the user input panel for **ImgMod24** is shown in Figure 2. A sample of the input panel for **ImgMod12** is shown in Figure 13. This makes it possible for the user to provide different input values each time the image-processing method is rerun. To rerun the image-processing method, type the new value into the text field and press the **Replot** button.

Discussion and Sample Code

Before getting into the details of the image-processing programs, I am going to briefly cover the two programs named **ImgMaker01** and **ImgMaker02**. These two programs are utility programs that I wrote to produce special jpg image files. I will use those files to illustrate certain key aspects of the two image-processing programs.

The program named ImgMaker01

The program named **ImgMaker01** is shown in Listing 20. The purpose of this program is to write an output jpg file named **junk.jpg** containing a white square centered in a square black image as shown in Figure 3.



The size of the square and the image

The length of the sides of the image and the length of the sides of the white square are provided by the user as command line parameters. If the user doesn't provide these values, the default size of the image is 31 pixels on each side and the default size of the white square is 9 pixels on each side.

Usage information

To run this program, enter the following command at the command-line prompt

```
java ImageMaker01 ImageSize SquareSize
```

where:

- ImageSize is the number of pixels on the side of the square image.
- SquareSize is the number of pixels on the side of the white square centered in the image.

Color values

The red, green, and blue values of the pixels in the white square are all 255.

The value of the alpha byte for all pixels is set to 255. *(See later note regarding the writing of the jpg file.)*

All red, green, and blue pixel values outside the white square are zero.

> *(Note that when these values are encoded into the jpg file and later read into another program, some of the values may be found to exhibit small errors. Apparently this is the result of encoding and later decoding the data in the jpg file.)*

The alpha byte

This program cannot handle alpha bytes with different values when writing the file. Rather, the program writes the three color bytes into the output file, apparently setting all of the alpha bytes to 255.

References

This file writing capability is based on information obtained from the following websites:

- https://jaistuff.dev.java.net/data.html
- https://jaistuff.dev.java.net/Code/data/CreateRGBImage.java

Program code

The program contains two static methods that:

- Generate the pixel values
- Encode those pixel values into the output jpg file named **junk.jpg**

The names of the two methods are:

- createThreeDImage
- writeImageFile

The createThreeDImage method

The code in this method is relatively straightforward and shouldn't require much of an explanation.

The method stores the pixel data for the white square into a 3D array of type:

int[row][column][depth].

The first two dimensions of the array correspond to the rows and columns of pixels in the image.

The third dimension always has a value of 4 and contains the following information by index value:

- 0 alpha value *(not set within the program)*
- 1 red value
- 2 green value
- 3 blue value

Note that these values are stored as type **int** rather than type **unsigned byte** which is the format of pixel data in an image. The values are converted to type **unsigned byte** during the writing of the jpg file.

The writeImage method

The code in the second method is not straightforward at all. However since the purpose of this lesson is to concentrate on processing image files rather than writing image files, I am simply going to refer you to the two URLs listed above for an explanation of that code.

The program was tested using SDK 1.4.2 under WinXP.

The program named ImgMaker02

The program named **ImgMaker02** is shown in Listing 21. The purpose of this program is to write an output jpg file named **junk.jpg** containing a single white pixel centered in a square black image as shown in figure 4. Images like this will be used for impulse testing the two image-processing programs to be discussed later in this lesson.

This program uses the same method for creating the jpg file that was discussed with regard to the program named **ImgMaker01**.

Furthermore, the data-generation portion of this program is even simpler than the data-generation portion of the program named **ImgMod01**. Therefore, I won't discuss this program further other than to tell you that you can run the program by entering the following at the command-line prompt

```
java ImageMaker02 ImageSize
```

where:

- ImageSize is the number of pixels on the side of the square image.

The program named ImgMod24

That brings us to the first of the two image-processing programs that I will explain in this lesson. This program is named **ImgMod24**. Before getting into the details of this program, however, I want to explain certain aspects of convolution.

Convolution is a linear process

I explained in the lesson entitled Convolution and Frequency Filtering in Java that convolution is a linear process. Among other things, this means that superposition holds. It is possible to reverse the order of certain operations without changing the overall results.

A convolution example

For example, assume that I have a time series that contains high-frequency components that I would like to suppress. I can accomplish that by convolving the time series with a low-pass convolution filter that will suppress the high-frequency components.

Suppose that after applying the convolution operator once to the time series, I conclude that there is still too much energy in the high-frequency area. There is nothing to stop me from simply applying the low-pass convolution filter again to further suppress the high-frequency components.

Now suppose that I know in advance that one pass of the convolution filter won't do the job and I would like to create a different convolution filter that will do the job in a single pass. One way to do this is to convolve the convolution filter with itself to produce an output that is a new convolution filter. I can then apply this new convolution filter to the time series attaining acceptable high-frequency suppression with a single pass. In fact, the results will be identical to the results obtained by applying the original convolution filter twice.

Which approach would be preferable?

Both approaches will provide the same results. I can either apply the convolution operator to the time series twice in succession, or I can apply the convolution operator to itself and convolve the output from that convolution process with the time series once.

Therefore, my evaluation as to which approach is best must be based on something other than the frequency content of the time series following the application of the convolution filter.

Required computing resource as an evaluation criteria

One evaluation criteria might be the amount of computing resource that is required to accomplish each approach.

To follow up on the issue of required computing resource, assume that I have two convolution filters. The first is a three-point filter having the following coefficient values:

1, 1, 1

The second is a five-point filter having the following coefficient values:

1, 2, 3, 2, 1

Which of these two convolution filters would require the greatest computing resource to apply? The answer is simple. The second filter would require the greatest computing resource for two reasons:

- The second filter has more coefficients and therefore requires more computations.
- The second filter requires multiplication by values other than unity.

The cost of multiplication

With some systems, the second reason is much more important than the first reason. Although the computation of a convolution output value always requires computing the sum of the products of the filter coefficient values and the data values, when all of the filter values are 1, the multiplication step can be skipped.

On many systems, multiplication is very expensive in terms of computer resources. Therefore, the requirement to do multiplication can be much more significant in terms of required computer resources than the number of points in the convolution filter.

Convolve the first filter with itself

Now, take out a piece of paper and convolve the first filter given above with itself. What did you get? If I did it correctly, I got a result that is the second convolution filter given above. Therefore, convolving the time series with the first filter twice in succession will produce the same result as convolving the time series with the second filter once.

Might be more efficient

Because all of the coefficients in the first filter have a value of 1, you should be able to write a special convolution algorithm that doesn't do any multiplication.

(That isn't possible with the second filter because it contains values other than 1.)

As a result, you may be able to write an algorithm that will convolve the time series with the first filter twice in succession and still require less computing resource than the algorithm to convolve the time series with the second filter only once.

Now convolve one more time

Now convolve the second filter with itself. If I did the arithmetic correctly, this results in a filter containing the following nine coefficient values:

1, 4, 10, 16, 19, 16, 10, 4, 1

Convolving this filter with the time series once would produce the same result that would be produced by convolving the first filter with the time series four times in succession. However, that isn't my main point in having you do this.

Approaches a Gaussian

If you plot this new filter in Cartesian coordinates, you might notice that the shape of the curve is tending towards a typical *bell shaped* or *Gaussian* curve.

Without getting into the technical details, a convolution operator having a Gaussian shape has some very interesting properties in digital signal processing *(DSP)*, so that is something that we might be interested in.

If we begin with a flat convolution filter and successively convolve it with itself, the resulting convolution filter will more and more closely approximate a Gaussian shape.

Similarly, because the convolution process is a linear process and superposition holds, if we successively convolve a time series with a flat convolution filter, the ultimate result will be the same as convolving that time series with a convolution filter having a Gaussian shape. Unlike with the actual Gaussian filter, however, we can write a convolution algorithm for a flat filter that doesn't require any multiplications *(except for possibly scaling or normalizing the final result).* Convolving multiple times in succession with a flat filter may require less computer resource than convolving with a Gaussian filter only once.

What about a 3D Gaussian filter?

I briefly described the 3D image convolution process in an earlier section. I will get into the detailed code that accomplishes that process later. Right now, I want to show you what happens when I successively convolve an image with a flat convolution filter consisting of an array of nine points all having the same value.

The top black image in Figure 5 contains a single white pixel, containing equal contributions of red, green, and blue. The color values for each of the three colors for this pixel are 255. The color values for all of the other pixels in the image are 0. Thus, this is what we would refer to as an impulse in DSP involving sampled time series.

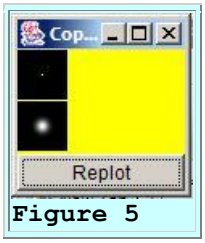


Figure 5

The result of multiple successive convolutions

The bottom image in Figure 5 shows the result of convolving the top image ten times in succession with the nine-point flat convolution filter whose values are shown in Figure 6. As you can see, the white color belonging to the single pixel in the center gets spread into the adjacent pixels. Not only did spreading occur, but the output is brightest in the center. The white color gradually progresses through grey to black as the distance from the center increases.

```
1 1 1
1 1 1
1 1 1
```
Figure 6

The output in numeric terms

Figure 7 shows the actual values that are displayed by the bottom image in Figure 5. *(These are the red color values only, but all three color values are the same for every pixel.)* The values

shown are the only non-zero values in the image. All the other pixel values in the image have a value of 0 and appear black in the bottom image of Figure 5.

```
                    1  2  4  4  4  2  1
              1  3  7 10 14 15 14 10  7  3  1
           1  4 10 19 28 36 39 36 28 19 10  4  1
        3 10 22 39 58 73 78 73 58 39 22 10  3
     1  7  9 39 68 99 123 132 123 99 68 39 19  7  1
  2 10 28 58 99 143 178 191 178 143 99 58 28 10  2
4 14 36 73 123 178 220 237 220 178 123 73 36 14  4
4 15 39 78 132 191 237 255 237 191 132 78 39 15  4
4 14 36 73 123 178 220 237 220 178 123 73 36 14  4
  2 10 28 58 99 143 178 191 178 143 99 58 28 10  2
     1  7 19 39 68 99 123 132 123 99 68 39 19  7  1
        3 10 22 39 58 73 78 73 58 39 22 10  3
           1  4 10 19 28 36 39 36 28 19 10  4  1
              1  3  7 10 14 15 14 10  7  3  1
                    1  2  4  4  4  2  1
```

Figure 7

A Gaussian shape with a round footprint

The value of 255 shown at the center of Figure 7 represents the brightest point in the center of the bottom image of Figure 5. As you move away from that value at the center, the other values shown in Figure 7 represent the grey values shown in Figure 5. Ultimately the black, or zero values occur, but they are not shown in Figure 7.

> *(I promised earlier that I would explain what I meant by a Gaussian shape with a round footprint. Figure 7 illustrates a Gaussian shape with a nearly round footprint. If you were to use clay and build a 3D model of the values shown in Figure 7, it would be nearly round on the bottom and would look like a church bell with a nearly Gaussian shape.)*

Plot some points on an intersection

If you draw a line through the center point in Figure 7 and plot the values intersected by that line in Cartesian coordinates, you will see that the values describe a *bell shape* or *Gaussian* curve.

Symmetry

If you divide Figure 7 into four quadrants centered on the value of 255 at the center, you will see that the other values exhibit symmetry about each of the axes.

What does this mean?

This means that if you convolve this nine-point flat convolution filter with each of the values of the three color surfaces ten times in succession, every pixel will be modified in the manner shown in the bottom image in Figure 5. Each pixel will maintain the correct relative height and

will be spread into the adjacent pixels in the manner shown in Figure 5. The resulting picture will be the sum of those modified pixels.

Back to the stick man

This should explain why the stick man in the bottom image of Figure 1 appears softer and fuzzier than the stick man in the top image of Figure 1. In this case, the outline of the original stick man results from a series of pixels that have all zero color values. Therefore, those pixels appear to be black.

The white areas in Figure 1 represent pixels whose red, green, and blue color values are all 255. As a result, those pixels appear to be white.

Each of the pixels at the transition between white and black in the bottom image of Figure 1 was modified in a manner similar to the bottom image in Figure 5. This results in the apparent fuzziness of the stick man in Figure 1.

Explanation from a DSP viewpoint

Another explanation, from a DSP viewpoint, is that rapid transitions from black to white require color surfaces containing strong high-frequency components. The convolution process implemented by this program suppresses high-frequency components from the color surfaces. Therefore, rapid transitions from black to white are also eliminated.

Because the black areas that represent the stick man are so narrow, elimination of the rapid transitions from black to white tend to turn the black stick man into a grey stick man. There simply isn't enough space to go from white to black and back to white in the width of the stick man's body.

This is most apparent by comparing the stick man's face with the remainder of his body. The width of the black area representing the face is wider than the other parts of his body. Therefore, the face ended up blacker than the rest of the body.

Now for some code - ImgMod24

The program named **ImgMod24** is designed to allow a user to apply the flat nine-point convolution filter shown in Figure 6 repetitively to an input image. The number of times the convolution filter is applied is specified by the user via the control panel shown in Figure 2.

The user can experiment by entering different values into the text field in Figure 2 and then pressing the **Replot** button in Figure 1. Each time the **Replot** button is pressed, the old processing results are cleared out and the image is processed and displayed again according to the new value provided by the user.

The processImg method

The image-processing program must implement the interface named **ImgIntfc02**. A listing of that interface was provided in the earlier lesson entitled Processing Image Pixels using Java, Getting Started. That interface declares a single method with the following signature:

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

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

It's best to make and modify a copy

Normally the **processImg** method should make a copy of the incoming array and modify the copy rather than modifying the original. Then the method should return a reference to the processed copy of the three-dimensional pixel array.

The program named ImgMod24

This program allows for multiple successive convolutions using a fixed 3x3 flat convolution filter. The result approaches a Gaussian filter as more successive convolutions are performed.

The output is normalized

The program normalizes the output so that the largest color value in the output always matches the largest color value in the input. This may or may not be desirable depending on the circumstances.

Driven by ImgMod02a

This program is designed to be driven by the program named **ImgMod02a**. Enter the following at the command line to run this program:

```
java ImgMod02a ImgMod24 ImagePathAndFileName
```

A low-pass filter

As mentioned above, this is a low-pass filter that suppresses high frequency components in color surfaces described by an array of color values.

The algorithm

The program treats each color surface separately from the others. During each convolution pass, the program adds all of the color values for each color surface within the area covered by the 3x3 filter. The sum of those values constitutes one value in the output color surface.

Then it moves to the next registration point and adds the pixel values then covered by the area. This process is continued until all of the values in the color surface have been processed.

When a convolution pass is complete, all of the color values in the output surface are scaled so that the peak color value in the output surface matches the peak color value in the input surface.

Special treatment at the edges

Each pixel belonging to the input color surface, except those at the outer edges of the surface, is used as a registration point. The pixels around the outer edges are not used as registration points because that would cause the area covered by the convolution filter to extend outside the input surface. The result of ignoring the outer edges of the input surface is shown by the black frame in the bottom image of Figure 1.

> *(If this were a production system, I would need to come up with a better way to handle the pixels at the edges rather than to just ignore them.)*

The visual effect

The visual effect of applying this filter to an image is to cause the image to go increasingly out of focus as the number of convolutions is increased. The effect is most obvious with images that have well-defined lines such as text characters. This is sometimes referred to as a blurring filter.

Why use a blurring filter?

One possible use of a blurring filter such as this is to reduce the visibility of age lines and wrinkles in a portrait of a human face, thus causing the person in the portrait to look somewhat younger.

Transparency

The transparency or alpha value of each pixel is preserved intact. If you don't see what you expect to see when you run this program with a particular image, it may be because your image contains transparent areas. This will be evidenced by the yellow background color of the canvas showing through the image.

Testing

This program was tested using SDK 1.4.2 and WinXP

The Graphical User Interface

The program provides the GUI control panel shown in Figure 2, which allows the user to enter a new value to specify the number of times to apply the convolution filter. To use this feature, simply type a new integer value into the text field and press the **Replot** button at the bottom of the main display frame shown in Figure 1.

No need to press the Enter key

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

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

Will discuss in fragments

I will break the program down and discuss it in fragments. A complete listing of the program is provided in Listing 22 near the end of the lesson.

The beginning of the class definition, including the declaration of some instance variables is shown in Listing 1.

```
class ImgMod24 extends Frame
implements

ImgIntfc02{

  int numberConvolutions;
  String inputData;//Obtained via the
TextField
  TextField input;//User input field

Listing 1
```

As is the case with all classes that are intended to be run under control of the program named **ImgMod02a**, this class implements the interface named **ImgIntfc02**. This in turn requires the class to define the method named **processImg**, which will be discussed shortly.

The constructor

The constructor is shown in Listing 2. The only purpose of the constructor is to create the control panel GUI shown in Figure 2. The code in the constructor is straightforward and should not require further discussion.

```
  ImgMod24(){//constructor
    setLayout(new FlowLayout());

    Label instructions = new Label(
                "Number of
convolutions/replot.");
    add(instructions);

    input = new TextField("1",5);
    add(input);
```

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

Listing 2
```

The processImg method

The **processImg** method, which is declared in the interface named **ImgInfc02**, begins in Listing 3.

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

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

    //Get numberConvolutions value
from the
    // TextField
    numberConvolutions =
Integer.parseInt(

input.getText());

Listing 3
```

The **processImg** method applies the convolution filter to the incoming 3D array of pixel data and returns a normalized filtered 3D array of pixel data. The output array is normalized such that the peak output color value matches the peak input color value.

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

A working copy of the 3D data

The code in Listing 4 makes a working copy of the incoming 3D array to avoid making permanent changes to the original image data. It also gets and saves the peak input color value for use in normalization later on.

```
    int inputPeak = 0;
    int colorValue = 0;
```

```
    int[][][] working3D =
                      new
int[imgRows][imgCols][4];
    for(int row = 0;row <
imgRows;row++){
      for(int col = 0;col <
imgCols;col++){
        working3D[row][col][0] =

threeDPix[row][col][0];
        colorValue =
threeDPix[row][col][1];
        working3D[row][col][1] =
colorValue;
        if(colorValue > inputPeak){
          inputPeak = colorValue;
        }//end if

        colorValue =
threeDPix[row][col][2];
        working3D[row][col][2] =
colorValue;
        if(colorValue > inputPeak){
          inputPeak = colorValue;
        }//end if

        colorValue =
threeDPix[row][col][3];
        working3D[row][col][3] =
colorValue;
        if(colorValue > inputPeak){
          inputPeak = colorValue;
        }//end if

      }//end inner loop
    }//end outer loop
    System.out.println(
                      "inputPeak = " +
inputPeak);
```

Listing 4

Miscellaneous preparation operations

The code in Listing 5 creates an empty output array of the same size as the incoming array. Then it copies all of the alpha or transparency values from the input array to the output array. No processing is performed on the alpha values.

```
    //Create an empty output array of
the same
    // size as the incoming array.
    int[][][] output =
                      new
int[imgRows][imgCols][4];
```

```
    //Copy all alpha values from input
to output.
    for(int row = 0;row <
imgRows;row++){
      for(int col = 0;col <
imgCols;col++){
        output[row][col][0] =

working3D[row][col][0];
      }//end inner loop
    }//end outer loop

Listing 5
```

The convolution operation

The convolution operation begins in Listing 6. This operation uses three nested **for** loops to treat each pixel *(other than those along the edges of the image)* as a registration point, and to perform the two-dimensional convolution using a *shift-sum-scale* approach. There is no multiplication required between convolution operator values and surface values.

> *(This algorithm is somewhat different from and probably more efficient than the algorithm used in the program named **ImgMod12** to be discussed later in this lesson. It is also simpler. However, this algorithm is also less flexible in terms of the shapes of the convolution filters that can be applied.)*

```
    //Perform the convolution one or
more times
    // in succession
    for(int cnt = 0;
                  cnt <
numberConvolutions;cnt++){
      try{
        //Iterate on each pixel as a
registration
        // point.
        for(int row = 0 + 1;row <
imgRows - 2;

row++){
          for(int col = 0 + 1;
                        col < imgCols
- 2;col++){

Listing 6
```

The three nested for loops

The convolution operation uses an outer loop to control the number of times the convolution operator is successively applied to the image.

The two inner loops iterate on the number of rows and the number of columns contained in the image to perform one convolution pass.

Listing 6 shows the setup code for the three nested **for** loops.

Calculate the red sum

Figure 7 shows the calculation that is performed to calculate the red output value for each input registration point during one convolution pass. Once again, note that there are no multiplications required. This is because the values of all the convolution operator coefficients are 1.

```
            int redSum =
                  working3D[row -
1][col - 1][1] +
                  working3D[row -
1][col - 0][1] +
                  working3D[row -
1][col + 1][1] +
                  working3D[row -
0][col - 1][1] +
                  working3D[row -
0][col - 0][1] +
                  working3D[row -
0][col + 1][1] +
                  working3D[row +
1][col - 1][1] +
                  working3D[row +
1][col - 0][1] +
                  working3D[row +
1][col + 1][1];
```

Listing 7

If you examine Listing 7 carefully, you will see that the calculation simply involves adding the nine input values centered on the registration point to produce the output value for that registration point.

Calculate the green and blue sums

Listing 22 near the end of the lesson shows two additional blocks of code, almost identical to the code in Listing 7. These blocks of code are used to calculate the green and blue sums. Because of the similarity of the code, I didn't include that code in this discussion of code fragments.

Store the sums in the output image

The code in Listing 8 stores the red, green, and blue sums in the output image for each registration point.

```
            output[row][col][1] =
```

```
redSum;
            output[row][col][2] =
greenSum;
            output[row][col][3] =
blueSum;

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

      }catch(Exception e){
        e.printStackTrace();
      }//end catch

Listing 8
```

Listing 8 also shows the ends of the two inner **for** loops that iterate on rows and columns.

Get output peak value for normalization

The code in listing 9 scans the red, green, and blue color values in the output image to get and save the peak color value. This value will be used to normalize the output image to the same peak value as the input image.

```
      int outputPeak = 0;
      for(int row = 0;row <
imgRows;row++){
        for(int col = 0;col <
imgCols;col++){
          if(output[row][col][1] >
outputPeak){
            outputPeak =
output[row][col][1];
          }//end if
          if(output[row][col][2] >
outputPeak){
            outputPeak =
output[row][col][2];
          }//end if
          if(output[row][col][3] >
outputPeak){
            outputPeak =
output[row][col][3];
          }//end if
        }//end inner loop
      }//end outer loop
      //System.out.println(
      //           "outputPeak = " +
outputPeak);

Listing 9
```

Normalize the peak value

The code in Listing 10 uses the two peak values that were saved earlier to scale all of the values in the output image to make the peak color value in the output image equal to the peak color value in the input image.

```
      double outputScale =

((double)inputPeak)/outputPeak;
      for(int row = 0;row <
imgRows;row++){
        for(int col = 0;col <
imgCols;col++){
          output[row][col][1] =

(int)(output[row][col][1]*

outputScale);
          output[row][col][2] =

(int)(output[row][col][2]*

outputScale);
          output[row][col][3] =

(int)(output[row][col][3]*

outputScale);
        }//end inner loop
      }//end outer loop

Listing 10
```

Reprocess or return?

At this point, a decision must be made to either loop back and apply the convolution filter again to the previously processed data, or to return the processed data to the program named **ImgMod02a**.

Copy output data to input array

In view of the possibility that it may be necessary to perform another convolution pass on the processed data, the code in Listing 11 copies the processed normalized output color data into the input working array. Then control returns to the top of the **for** loop where a decision is made to either process the data again, or to break out of the loop and return to **ImgMod02a**.

> *(An improvement in structure could be made at this point to prevent the unnecessary copying of the data at the end of the final convolution pass.)*

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

```
          working3D[row][col][1] =

output[row][col][1];
          working3D[row][col][2] =

output[row][col][2];
          working3D[row][col][3] =

output[row][col][3];
        }//end inner loop
      }//end outer loop
    }//end for loop on
numberConvolutions

Listing 11
```

Return the processed image

Listing 12 shows the code that is executed when all the processing has been completed and it is time to return the processed image to the program named **ImgMod02a** for display.

```
    System.out.println("Processing
Done");
    return output;
  }//end processImg method
}//end class ImgMod24

Listing 12
```

Listing 12 also shows the end of the **processImg** method and the end of the **ImgMod24** class.

Some more image-processing examples

Before we finish our discussion of this program, let's look at a few more image-processing examples.

Figure 8 shows the result of making ten convolution passes on an image containing a white square. This example clearly illustrates the manner in which this processing technique softens the hard transitions between colors.

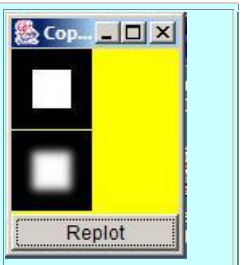


Figure 8 with 10 convolution

passes

Edge detection

In a future lesson I will show you another convolution technique that emphases rather than softens the edges of transitions between colors. Convolution is used in both cases. The only difference is the convolution operator that is used.

A natural example

Up to this point, all of the results that I have shown you have been based on artificial images, so to speak. They were not images taken from nature. Figure 9 shows the result of making ten convolution passes on an image taken from a digital photograph at an aquarium.

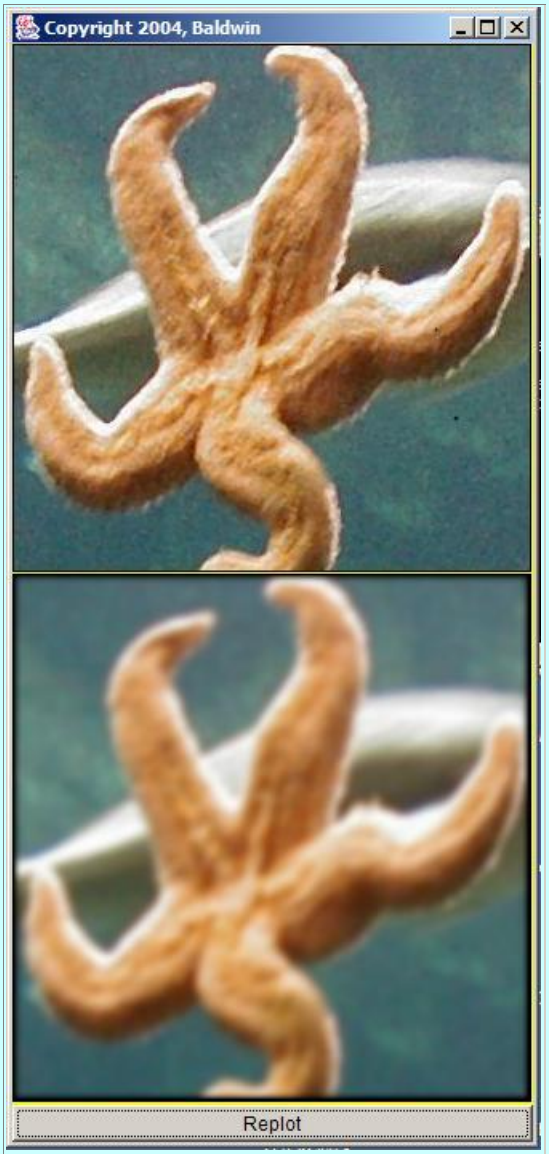


Figure 9 with 10 convolutions

Note that Figure 9 is not intended to improve the image. It is intended simply to show you the result of convolution with this particular operator.

Application to text characters

The application of a smoothing operator is most apparent for situations where there are well-defined lines, such as in text. This is illustrated in Figure 10, which shows the result of making only one convolution pass with the flat 3x3 operator on an image containing text.

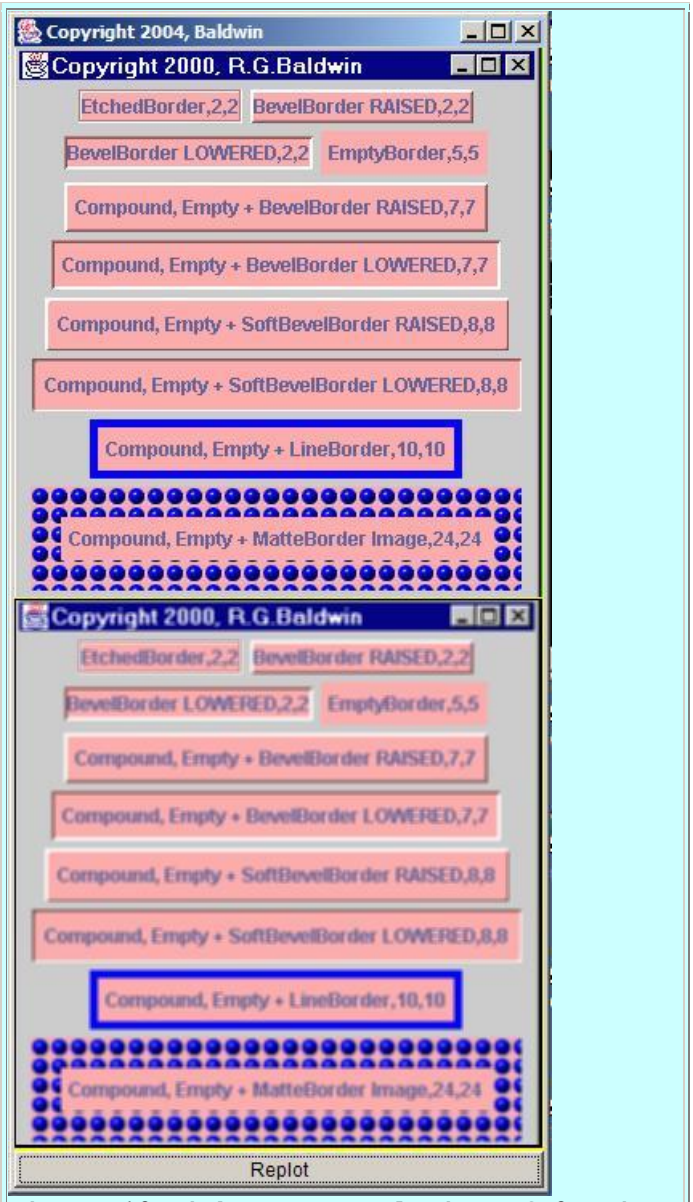


Figure 10 with one convolution of 9 points

As you can see, this causes the transitions between colors to become less well defined. This has the effect of blurring the characters and the lines.

Additional blurring

Figure 11 shows the result of making ten convolution passes on the same image. As you can see, this caused the text to become almost totally unreadable.

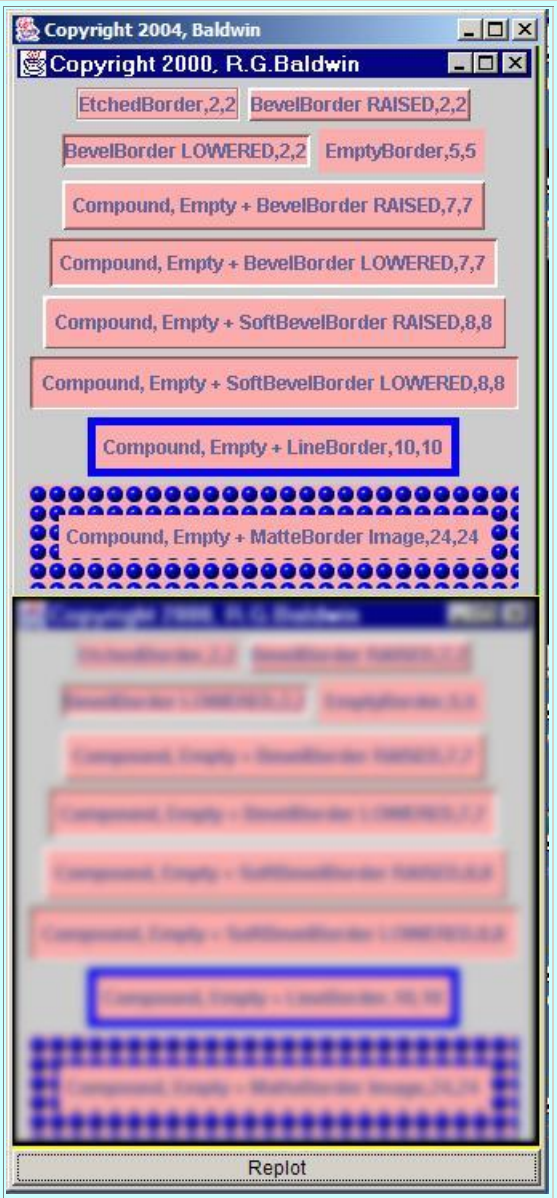


Figure 11 with 10 convolutions

A different approach

Next, I am going to discuss the program named **ImgMod12**, which uses a completely different approach to the use of convolution for smoothing. After discussing that program, I will show you some additional image-processing examples and use them to compare the two approaches.

One common situation

There is one situation in which the two approaches are the same. Making a single convolution pass with **ImageMod24** is equivalent to processing with **ImgMod12** using a 3x3 convolution operator. This is the situation illustrated in Figure 10.

Making ten convolution passes using **ImgMod24** is roughly equivalent to using a Gaussian filter with a nearly round footprint about fifteen pixels in diameter *(see Figure 7).*

The program named ImgMod12

The program named **ImgMod12** applies a flat convolution filter to an input image. The user is allowed to control the size and to some extent, the 2D shape of the filter, but it is always flat regardless of user input. Because of the additional requirement for control code to accommodate the user input, the code is more complex than the code in the previously-discussed program named **ImgMod24.**

Sample output from ImgMod12

Figure 12 shows the output from this program for the one case where the behavior of this program matches the behavior of the program named **ImgMod24**. This is the case where both programs apply a square 3x3 flat filter. This is the startup case for **ImgMod24** and is one of the selectable cases for ImgMod12.

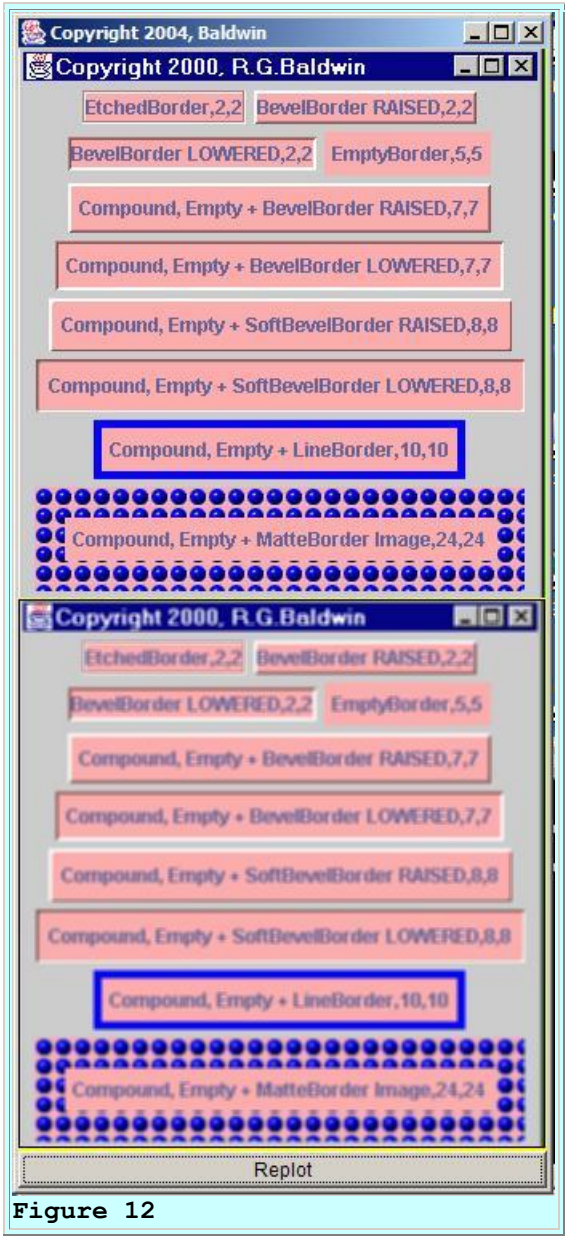


Figure 12

The bottom image in Figure 12 should compare favorably with the bottom image in Figure 10, which was produced by the program named **ImgMod24**.

The interactive control panel for ImgMod12

Figure 13 shows the interactive control panel for **ImgMod12**, which allows the user to specify the area in sample points for the flat convolution filter that is to be applied to the input image.

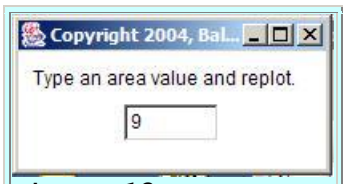


Figure 13

Runs under control of ImgMod02a

The program named **ImgMod12** is designed to be driven by the program named **ImgMod02a**. Enter the following at the command line prompt to run this program:

```
java ImgMod02a ImgMod12 ImageFileName
```

This program illustrates the use of area convolution filtering to blur or soften an image.

Display format

The program displays two frames on the screen. The large frame on the left shows the original image at the top and the filtered image at the bottom. That frame has a button labeled **Replot** at the very bottom.

The small frame on the right is the interactive control pane shown in Figure 13. It contains a **TextField** for user input.

Interactive control panel

When the program starts running, this **TextField** displays the size of the default convolution area in pixels. To modify the convolution area, type an integer value into the **TextField** and click the **Replot** button. The new filter will be applied to the image and the filtered image will be displayed.

Shape and size of convolution filters

The program supports non-square convolution area values of 1, 2, 3, 4, 6, and 8 pixels. The shape of the convolution area is shown as a grid of X characters on the screen. Area values of 0, 5, and 7 are not supported.

In addition, the program supports all area values that are perfect squares beginning with an area value of 4 pixels. For area values greater than 9, the value entered by the user is automatically rounded to the nearest perfect square before processing takes place. For example, if the user enters 10, the actual area used for convolution will be a square with 3 pixels on each side. If the user enters 15, the area used for convolution will be a square with 4 pixels on each side. The convolution operator is a box with each coefficient having a value of 1. *(See discussion of normalization later.)*

Mechanics of convolution

This is a low-pass filter that suppresses high frequency changes in color values. The red, green, and blue color surfaces are treated separately. The program adds all of the pixel values for each color within the area covered by the filter and uses that value to produce an output point. Then the program moves to the next registration point and adds the pixel values that are contained in the area there.

Special treatment at the edges

Every pixel, except those in the outer edges of the image, is used as a registration point.

> *(The pixels around the outer edges are not used as registration points because that would cause the convolution area to extend outside the color surface)*

Normalization

Once the convolution process is finished, the output data is normalized such that the peak color value in the output matches the peak color value in the input. This may, or may not be appropriate depending on the circumstances. However, it does preserve the dynamic range of the display.

The visual effect

The visual effect of applying this filter is to cause the image to go increasingly out of focus as the size of the area is increased. The effect is most obvious with images that have well defined lines such as text characters.

Transparency is preserved

The transparency or alpha value of each pixel is preserved. If you don't see what you expect to see when you run this program with a particular image, it may be because your image contains transparent areas. This will be evidenced by the yellow background color of the canvas showing through the image.

Testing

The program was tested using SDK 1.4.2 and WinXP.

Will discuss in fragments

I will break the program named **ImgMod12** down and discuss it in fragments. Listing 13 shows the beginning of the class and the constructor.

```
class ImgMod12 extends Frame
implements

ImgIntfc02{
```

```
  int area;//The area value in pixels
  String inputData;//Obtained via the
TextField
  TextField input;//User input field

  ImgMod12(){//constructor
    setLayout(new FlowLayout());

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

    input = new TextField("2",5);
    add(input);

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

Listing 13

Once again, note that the class implements the interface named **ImgIntfc02** requiring the class to define the method named **processImg**.

The constructor simply creates the user input panel shown in Figure 13.

The processImg method

The **processImg** method begins in Listing 14. This method applies the convolution filter to the incoming 3D array of pixel data and returns a filtered 3D array of pixel data.

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

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

    //Get area value from the
TextField
    area =
Integer.parseInt(input.getText());

    //Create an empty output array of
```

```
the same
    // size as the incoming array.
    int[][][] output =
                        new
int[imgRows][imgCols][4];

    //Make a working copy of the 3D
array to
    // avoid making permanent changes
to the
    // original image data.  Get and
save the
    // maximum value along the way.
    int inputPeak = 0;
    int colorValue = 0;
    int[][][] working3D =
                        new
int[imgRows][imgCols][4];
    for(int row = 0;row <
imgRows;row++){
      for(int col = 0;col <
imgCols;col++){
        working3D[row][col][0] =

threeDPix[row][col][0];
        colorValue =
threeDPix[row][col][1];
        working3D[row][col][1] =
colorValue;
        if(colorValue > inputPeak){
          inputPeak = colorValue;
        }//end if

        colorValue =
threeDPix[row][col][2];
        working3D[row][col][2] =
colorValue;
        if(colorValue > inputPeak){
          inputPeak = colorValue;
        }//end if

        colorValue =
threeDPix[row][col][3];
        working3D[row][col][3] =
colorValue;
        if(colorValue > inputPeak){
          inputPeak = colorValue;
        }//end if

      }//end inner loop
    }//end outer loop
    System.out.println(
                       "inputPeak = " +
inputPeak);

    //Copy all alpha values from input
```

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

working3D[row][col][0];
      }//end inner loop
    }//end outer loop

Listing 14
```

The code in Listing 14 is very similar to the code discussed earlier for the program named **ImgMod24**, so there should be no need to repeat that discussion here.

Accumulators

Listing 15 declares three variables that are used to accumulate the products of the pixel values and the convolution filter coefficients

> *(Note however that because the values of all of the convolution filter coefficients are 1, no actual multiplication is required. The program would probably run much more slowly if it were actually necessary to multiply the pixel values by the filter coefficients.)*

```
    int redSum = 0;
    int greenSum = 0;
    int blueSum = 0;

Listing 15
```

Control variables

Listing 16 declares a large number of variables that are used for control purposes while performing the convolution operation.

```
    int rowNo = 0;
    int colNo = 0;
    int row = 0;
    int col = 0;
    int firstRow = 0;
    int lastRow = 0;
    int firstCol = 0;
    int lastCol = 0;
    int minusRow = 0;
    int plusRow = 0;
    int minusCol = 0;
    int plusCol = 0;
```

Listing 16

Setting the control variables

Listing 17 contains a **switch** statement that is used to set the control variables listed above for area values of 1, 2, 3, 4, 6, and 8 on an individual area basis. Area values of 5 and 7 are not supported. Area values of 9 and greater default to the nearest perfect square, such as 9, 16, 25, 36, etc.

```
    switch(area){
      case 0:
        System.out.println(
                    "Area value 0 not
supported");
        break;
      case 1://A single pixel
reproduces image
        firstRow = 0;
        lastRow = imgRows;
        firstCol = 0;
        lastCol = imgCols;
        minusRow = 0;
        plusRow = 0;
        minusCol = 0;
        plusCol = 0;
      break;
      case 2://Two pixels in a row
        firstRow = 0;
        lastRow = imgRows;
        firstCol = 1;
        lastCol = imgCols;
        minusRow = 0;
        plusRow = 0;
        minusCol = 1;
        plusCol = 0;
      break;
      case 3://Three pixels in a row
        firstRow = 0;
        lastRow = imgRows;
        firstCol = 1;
        lastCol = imgCols - 1;
        minusRow = 0;
        plusRow = 0;
        minusCol = 1;
        plusCol = 1;
      break;
      case 4://Four pixels in a square
        firstRow = 1;
        lastRow = imgRows;
        firstCol = 1;
        lastCol = imgCols;
        minusRow = 1;
        plusRow = 0;
        minusCol = 1;
```

```
        plusCol = 0;
      break;
      case 5:
        System.out.println(
                    "Area value 5 not
supported");
      break;
      case 6://Two rows of 3 pixels
        firstRow = 1;
        lastRow = imgRows;
        firstCol = 1;
        lastCol = imgCols - 1;
        minusRow = 1;
        plusRow = 0;
        minusCol = 1;
        plusCol = 1;
      break;
      case 7:
        System.out.println(
                    "Area value 7 not
supported");
      break;
      case 8://Two rows of 4
        firstRow = 1;
        lastRow = imgRows;
        firstCol = 2;
        lastCol = imgCols - 1;
        minusRow = 1;
        plusRow = 0;
        minusCol = 2;
        plusCol = 1;
      break;
      //Default to nearest perfect
square for
      // area values greater than 8.
      default:
        //Get the side of the square
area,
        // rounded to the nearest
square.
        double dSide =
Math.sqrt(area);
        int side =
(int)Math.round(dSide);

        //Set the area value to the
nearest
        // perfect square.  This is
necessary
        // because it is used to scale
the
        // accumulated values later.
        area = side*side;

        //Because a square area with
an even
```

```
        // number of pixels on a side
doesn't
        // have a pixel at the center,
it must
        // be treated differently from
a square
        // area with an odd number of
pixels on a
        // side.  For the even case,
the area
        // above and to the left of
the
        // registration point is
slightly greater
        // than the area below and to
the right.
        if(side%2 == 0){//side is even
          firstRow = side/2;
          lastRow = imgRows - side/2 +
1;
          firstCol = side/2;
          lastCol = imgCols - side/2 +
1;
          minusRow = side/2;
          plusRow = side/2 - 1;
          minusCol = side/2;
          plusCol = side/2 -1;
        }else{//side is odd
          firstRow = side/2;
          lastRow = imgRows - side/2;
          firstCol = side/2;
          lastCol = imgCols - side/2;
          minusRow = side/2;
          plusRow = side/2;
          minusCol = side/2;
          plusCol = side/2;
        }//end else
    }//end switch statement

Listing 17
```

The comments in Listing 17 should be sufficient to make the code self-explanatory.

Perform the convolution

The code in Listing 18 uses nested **for** loops to treat each pixel *(other than those along the edges of the image)* as registration points and to perform the two-dimensional convolution based on those registration points.

```
    try{
      //First iterate on each pixel as
a
```

```
      // registration point.
      for(row = firstRow;row <
lastRow;row++){
        for(col = firstCol;col <
lastCol;col++){

          //Now use the registration
point as a
          // base and iterate on the
pixels
          // contained within the area
covered by
          // the convolution filter.
Display a
          // grid of X characters on
the screen
          // showing the shape of the
area
          // covered by the
convolution filter.
          // Display the grid only
once while
          // processing the first
registration
          // point.
          for(rowNo = row - minusRow;
                 rowNo <= row +
plusRow;rowNo++){

            //Start a new line in the
grid of X
            // characters.
            if((row == firstRow)
                           && (col ==
firstCol)){
              System.out.println();
            }//end if

            for(colNo = col -
minusCol;
                colNo <= col +
plusCol;colNo++){

              //Display the next X in
the grid of
              // X characters.
              if((row == firstRow)
                           && (col ==
firstCol)){
                System.out.print("X");
              }//end if

              //Accumulate the pixel
values
              // multiplied by the
coefficient
```

```
              // values in the
convolution
              // filter.  Note that
all
              // coefficients have a
value of 1.
              // The accumulated value
will later
              // be divided by the
area, causing
              // the effective values
of the
              // coefficients to be
the
              // reciprocal of the
area.
              redSum +=

working3D[rowNo][colNo][1];
              greenSum +=

working3D[rowNo][colNo][2];
              blueSum +=

working3D[rowNo][colNo][3];
            }//end for loop on y
          }//end for loop on x

          //Store the accumlator
values in the
          // output array.
          output[row][col][1] =
redSum;
          output[row][col][2] =
greenSum;
          output[row][col][3] =
blueSum;

          //Clear the accumulators in
preparation
          // for processing the next
registration
          // point.
          redSum = 0;
          greenSum = 0;
          blueSum = 0;

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

    }catch(Exception e){
      e.printStackTrace();
    }//end catch
```

Listing 18

As you can see, the code in Listing 18 is much more complex than the code that performs the convolution for the program named **ImgMod24**. This increased complexity results from the fact that this program is much more flexible in terms of the size and shape of the convolution filter.

Normalize the data and return

The code in Listing 19 normalizes the output data to cause the peak color value in the output to match the peak color value in the input. Then the method returns the output data to the program named **ImgMod02a** for display.

```
    //Normalize output peak value to
match
    // input peak value.
    //First get output peak value
    int outputPeak = 0;
    for(row = 0;row < imgRows;row++){
      for(col = 0;col <
imgCols;col++){
        if(output[row][col][1] >
outputPeak){
          outputPeak =
output[row][col][1];
        }//end if
        if(output[row][col][2] >
outputPeak){
          outputPeak =
output[row][col][2];
        }//end if
        if(output[row][col][3] >
outputPeak){
          outputPeak =
output[row][col][3];
        }//end if
      }//end inner loop
    }//end outer loop

    //Normalize to peak value
    double outputScale =

((double)inputPeak)/outputPeak;
    for(row = 0;row < imgRows;row++){
      for(col = 0;col <
imgCols;col++){
        output[row][col][1] =
           (int)(output[row][col][1]*

outputScale);
        output[row][col][2] =
           (int)(output[row][col][2]*

outputScale);
        output[row][col][3] =
           (int)(output[row][col][3]*
```

```
outputScale);
      }//end inner loop
    }//end outer loop

    //Return a reference to the array
containing
    // the filtered pixels.
    return output;
  }//end processImg method
}//end class ImgMod12
```

Listing 19

The code in Listing 19 is very similar to the corresponding code discussed earlier for the program named **ImgMod24**. Therefore, I won't discuss it further here.

Note that Listing 19 also signals the end of the **processImg** method and the end of the **ImgMod12** class.

Some more examples from ImgMod12

Let's look at the output from some more examples. First consider the output shown in Figure 14 and compare it with the output from the program named **ImgMod24** shown earlier in Figure 5.

Figure 14

These two figures compare the impulse responses of the two convolution processes for convolution filters having approximately the same area.

The areas of the two filters

If you consider the footprint of the Gaussian filter shown in Figure 7 to be a perfect circle, the area of the circle is approximately 176 pixels. The output shown in Figure 14 was produced by specifying a 2D convolution area for **ImgMod12** to be 169 pixels. In particular, this is a square flat convolution filter that is 13 pixels on each side.

Contribution from pixels some distance from the center

For the Gaussian filter, the output produced for each registration point consists of the value at the registration point plus a decreasing contribution from pixels located within the nearly round footprint but at greater distances from the registration point.

For the flat filter used in **ImgMod12**, the output value for a given registration point consists of equal contributions of all the pixels contained within the rectangular or square footprint. Thus, for footprints of approximately the same area, the flat filter used in **ImgMod12** is a much harsher filter than the filter in **ImgMod24** that decays with distance from the center.

A much harsher filter

The fact that the filter in **ImgMod12** is much harsher for the same footprint area can be illustrated by comparing Figure 15 with Figure 9. Figure 9 was produced by **ImgMod24** and Figure 15 was produced by **ImgMod12**.

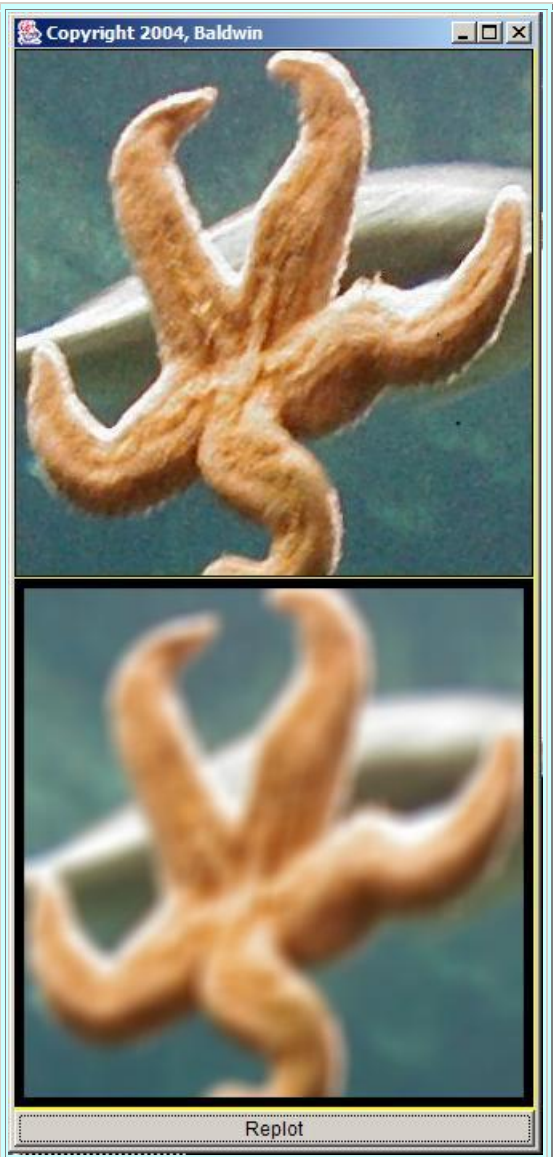


Figure 15

The total area encompassed by the footprints of the two filters was approximately the same *(169 for ImgMod12 and 176 for ImgMod24).* However, the blurring in Figure 15 was much more substantial than in Figure 9.

Neither good nor bad

This is not intended to indicate that one approach is better than the other. It is simply intended to show that the two approaches produce different results for the same total area encompassed within the footprint of the convolution filter. If your needs are such that you would prefer that the contribution of the pixels *(to the output)* decrease with distance from the registration point, then the Gaussian approach is probably best. On the other hand, if you need an equal contribution from all the pixels within the footprint, then the flat filter is probably best.

Interpretation of Results

The convolution process always produces an output sample as a weighted summation of input samples. The shape of the convolution operator along with the values of the individual coefficients in the convolution operator determine which input samples will be used to produce the output sample, and how they will be weighted in the output. Different convolution operators can produce decidedly different results.

A low-pass filter

In DSP terms, the convolution filters used in this lesson are what we would call low-pass filters. That is, they suppress high-frequency components and preserve low-frequency components.

In order for an image to exhibit rapid changes in color, the color values in the image must include high-frequency components. Suppressing those high-frequency components causes the transitions between colors to be spread across more pixels, thus producing the softening or blurring of the images that you have seen in the examples in this lesson.

In future lessons, I will show you what happens to your image when you use a convolution filter that preserves high-frequency components and suppresses low-frequency components. In general, this will result in sharpening the image, and in the extreme case, causing the edges between color transitions to become very prominent.

Run the Program

I encourage you to copy, compile and run the following programs that are provided in this lesson:

- ImgMaker01
- ImgMaker02
- ImgMod12

- ImgMod24

Experiment with them, making changes and observing the results of your changes.

> *(Remember, you will also need to copy the program named **ImgMod02a** and the interface named **ImgIntfc02** from the earlier lessons entitled Processing Image Pixels Using Java: Controlling Contrast and Brightness and Processing Image Pixels using Java, Getting Started.)*

Test images

To replicate the output images shown in this lesson, you will need to use the same images as input. Some of those images can be created by running the programs named **ImgMaker01** and **ImgMaker02**.

The other images are provided below. Simply right-click on each of the images in Figures 16, 17, and 18, and save them on your disk. Then use them as input to the programs named **ImgMod12** and **ImgMod24**.

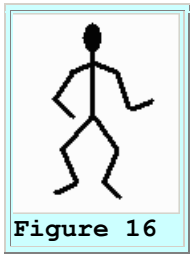


Figure 16

Figure 17

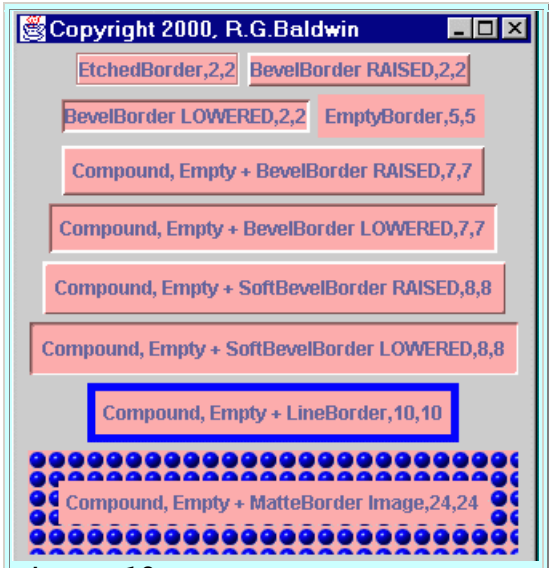


Figure 18

Modify a variety of images

If you search the Internet, you should be able to find lots of images that you can download and experiment with. Just remember, as explained in the lesson entitled Processing Image Pixels Using Java: Controlling Contrast and Brightness, if you download a gif image, it will probably contain a lot less color information than a comparable jpg image.

Have fun and learn

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

Summary

In this lesson, I showed you how to write programs that produce highly specialized jpg image files containing images that are very useful for testing image-processing programs.

I also showed you two different ways to perform convolution on an image to provide varying degrees of smoothing or blurring.

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:

- Deal with the effects of noise in an image.
- Sharpen all or part of an image.

- Perform edge detection on an image.
- Morph one image into another image.
- Rotate an image.
- Change the size of an image.
- Create a kaleidoscope of an image.
- Create a 3D or embossed effect with an image.
- Other special effects that I may dream up or discover while doing the background research for the lessons in this series.

Complete Program Listings

Complete listings of the programs discussed in this lesson are provided in Listing 20 through Listing 23. In order to use these programs, you will also need copies of the program named **ImgMod02a** and the interface named **ImgIntfc02** from the earlier lessons entitled Processing Image Pixels Using Java: Controlling Contrast and Brightness and Processing Image Pixels using Java, Getting Started.

A disclaimer

The programs that I am providing and explaining in this series of lessons are not intended to be used for high-volume production work. Numerous integrated image-processing programs are available for that purpose. In addition, the Java Advanced Imaging API *(JAI)* has a number of built-in special effects if you prefer to write your own production image-processing programs using Java.

The programs that I am providing 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 ImgMaker01.java
Copyright 2004, R.G.Baldwin

The purpose of this program is to write an output
jpg file named junk.jpg containing a white square
centered in a square black image.  The length of
the sides of the image and the length of the
sides of the white square are provided by the
user as command line parameters.  If the user
doesn't provide these values, the default size of
the image is 31 pixels on each side and the
default size of the white square is 9 pixels.

The output image files produced by this program
are very useful for testing and illustrating the
effects of 2D convolution.

Usage:
java ImageMaker01 ImageSize SquareSize
```

```
where:
ImageSize is the number of pixels on the side of
  the square image.
SquareSize is the number of pixels on the side of
  the white square centered in the image.

The red,green, and blue values of the pixels in
the white square are all 255.  The value of the
alpha byte for all pixels is set to 255. (See
later note regarding the writing of the jpg
file.)

All red, green, and blue pixel values outside the
white square are zero.  Note, however, that when
these values are written into the jpg file and
later read into another program, some of the
values may be found to exhibit small errors.
Apparently this is the result of encoding and
later decoding the data in the jpg file.

The program writes the image into a file named
junk.jpg.  Note that this program can't handle
alpha bytes with different values when writing
the file.  Rather, it writes the three color
bytes into the output file, apparently setting
the alpha byte to 255.

This file writing capability is based on
information obtained from the following web
sites:
https://jaistuff.dev.java.net/data.html
https://jaistuff.dev.java.net/Code/data/
  CreateRGBImage.java

The program stores the pixel data for the white
square into a 3D array of type:

int[row][column][depth].

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

0 alpha (not set within the program)
1 red
2 green
3 blue

Note that these values are stored as type int
rather than type unsigned byte which is the
format of pixel data in the an image.  The values
are converted to type unsigned byte during the
writing of the jpg file.

Tested using SDK 1.4.2 under WinXP.
```

```
**********************************************/

import java.awt.*;
import java.awt.image.*;
import javax.media.jai.*;

class ImgMaker01 extends Frame{

  public static void main(String[] args){
    int imgCols = 31;//default values
    int imgRows = 31;
    int whiteSquareSize = 9;

    if(args.length == 2){
      //Get size of image and size of white
      // square from command-line args.
      imgCols = Integer.parseInt(args[0]);
      imgRows = imgCols;
      whiteSquareSize =
                        Integer.parseInt(args[1]);
    }//end else
    int[][][] threeDPix = createThreeDImage(
                imgCols,imgRows,whiteSquareSize);
    writeImageFile(threeDPix,imgCols,imgRows);
  }//end main
  //----------------------------------------//

  static int[][][] createThreeDImage(
                             int imgCols,
                             int imgRows,
                             int whiteSquareSize){
    int[][][] temp3D =
                    new int[imgRows][imgCols][4];

    for(int col = imgCols/2 - whiteSquareSize/2;
            col < imgCols/2 + whiteSquareSize/2;
            col++){
      for(
         int row = imgRows/2 - whiteSquareSize/2;
         row < imgRows/2 + whiteSquareSize/2;
         row++){
        //Set values for red, green, and blue
        // colors in the white square.
        temp3D[row][col][1] = 255;//red
        temp3D[row][col][2] = 255;//green
        temp3D[row][col][3] = 255;//blue
      }//end inner for loop
    }//end outer for loop

    //Return the array of image data.
    return temp3D;
  }//end createThreeDImage
  //----------------------------------------//


  /*Write the image to a file named junk.jpg.
```

```
    Note that this program can't handle the alpha
    byte when writing the file.  Rather, the
    program only writes the three color bytes
    into the output file, apparently setting the
    alpha byte to 255. This file writing
    capability is based on information at:
    https://jaistuff.dev.java.net/data.html and
    https://jaistuff.dev.java.net/Code/data/
      CreateRGBImage.java
  */
  static void writeImageFile(int[][][] threeDPix,
                             int imgCols,
                             int imgRows){
    byte[] imageDataBytes = new byte[
                              imgCols*imgRows*3];
    int count = 0;

    for(int h = 0;h < imgRows;h++)
      for(int w = 0;w < imgCols;w++){
        //Rearrange the data into a one-
        // dimensional array of type byte. Note
        // that this array does not contain alpha
        // byte values.
        imageDataBytes[count+0] =
                         (byte)threeDPix[h][w][3];
        imageDataBytes[count+1] =
                         (byte)threeDPix[h][w][2];
        imageDataBytes[count+2] =
                         (byte)threeDPix[h][w][1];
        count += 3;
      }//end for loop

    // Create a Data Buffer from the values in
    // the single image array.
    DataBufferByte dbuffer = new DataBufferByte(
               imageDataBytes,imgCols*imgRows*3);

    // Create a pixel-interleaved data sample
    // model.
    SampleModel sampleModel =
       RasterFactory.
         createPixelInterleavedSampleModel(
                           DataBuffer.TYPE_BYTE,
                           imgCols,
                           imgRows,
                           3);

    // Create a compatible ColorModel.
    ColorModel colorModel =
       PlanarImage.createColorModel(sampleModel);
    // Create a WritableRaster.
    Raster raster =
           RasterFactory.createWritableRaster(
             sampleModel,dbuffer,new Point(0,0));

    // Create a TiledImage using the SampleModel.
```

```
    TiledImage tiledImage = new TiledImage(0,0,
                            imgCols,imgRows,0,0,
                            sampleModel,
                            colorModel);
    // Set the data of the tiled image to be the
    // raster.
    tiledImage.setData(raster);
    // Save the image in a file using one of the
    // overloaded versions of the create method.
    // Note that other file types can be written
    // by using a different value for the third
    // parameter as in the following:
    //JAI.create("filestore",tiledImage,
    //                        "junk.tif","TIFF");
    JAI.create("filestore",tiledImage,
                            "junk.jpg","JPEG");
  }//end writeImageFile
}//end ImgMaker01 class
```

Listing 20

```
/*File ImgMaker02.java
Copyright 2004, R.G.Baldwin

The purpose of this program is to write an output
jpg file named junk.jpg containing a single white
impulse centered in a square black image.  The
length of the sides of the image is provided by
the user as a command line parameter.  If the
user doesn't provide this value, the default size
of the image is 31 pixels on each side.

This program is useful for creating a jpg file
that can be used to get the 2D impulse response
of a 2D convolution filter.

Usage:
java ImageMaker01 ImageSize
where:
ImageSize is the number of pixels on the side of
  the square image.

The red,green, and blue values of the pixels in
the white impulse are all 255.  The value of the
alpha byte is 255 (see later discussion of the
alpha byte).

The program writes the image into a file named
junk.jpg.  Note that this program can't handle
alpha bytes with different values when writing
the file.  Rather, it writes the three color
bytes into the output file, apparently setting
the alpha byte to 255.

This file writing capability is based on
```

```
information obtained from the following web
sites:
https://jaistuff.dev.java.net/data.html
https://jaistuff.dev.java.net/Code/data/
  CreateRGBImage.java

The program stores the pixel data for the white
impulse into a 3D array of type:

int[row][column][depth].

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

0 alpha (not set in the program)
1 red
2 green
3 blue

Note that these values are stored as type int
rather than type unsigned byte which is the
format of pixel data in the an image.  The values
are converted to type unsigned byte during the
writing of the jpg file.

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

import java.awt.*;
import java.awt.image.*;
import javax.media.jai.*;

class ImgMaker02 extends Frame{

  public static void main(String[] args){
    int imgCols = 31;//default values
    int imgRows = 31;


    if(args.length == 2){
      //Get size of image from command-line args.
      imgCols = Integer.parseInt(args[0]);
      imgRows = imgCols;
    }//end else
    int[][][] threeDPix = createThreeDImage(
                                  imgCols,imgRows);
    writeImageFile(threeDPix,imgCols,imgRows);
  }//end main
  //----------------------------------------//

  static int[][][] createThreeDImage(int imgCols,
                                     int imgRows){
    int[][][] temp3D =
                    new int[imgRows][imgCols][4];
```

```
    int col = imgCols/2;
    int row = imgRows/2;

    //Set values for red, green, and
    // blue colors in the white impulse.
    temp3D[row][col][1] = 255;//red
    temp3D[row][col][2] = 255;//green
    temp3D[row][col][3] = 255;//blue

    //Return the array of image data.
    return temp3D;
  }//end createThreeDImage
  //----------------------------------------//

  /*Write the image to a file named junk.jpg.
    Note that this program can't handle the alpha
    byte when writing the file.  Rather, the
    program only writes the three color bytes
    into the output file, apparently setting the
    alpha byte to 255. This file writing
    capability is based on information at:
    https://jaistuff.dev.java.net/data.html and
    https://jaistuff.dev.java.net/Code/data/
      CreateRGBImage.java
  */
  static void writeImageFile(int[][][] threeDPix,
                             int imgCols,
                             int imgRows){
    byte[] imageDataBytes = new byte[
                              imgCols*imgRows*3];
    int count = 0;

    for(int h = 0;h < imgRows;h++)
      for(int w = 0;w < imgCols;w++){
        //Rearrange the data into a one-
        // dimensional array of type byte. Note
        // that this array does not contain alpha
        // byte values.
        imageDataBytes[count+0] =
                          (byte)threeDPix[h][w][3];
        imageDataBytes[count+1] =
                          (byte)threeDPix[h][w][2];
        imageDataBytes[count+2] =
                          (byte)threeDPix[h][w][1];
        count += 3;
      }//end for loop

    // Create a Data Buffer from the values in
    // the single image array.
    DataBufferByte dbuffer = new DataBufferByte(
               imageDataBytes,imgCols*imgRows*3);

    // Create a pixel-interleaved data sample
    // model.
    SampleModel sampleModel =
```

```
      RasterFactory.
        createPixelInterleavedSampleModel(
                            DataBuffer.TYPE_BYTE,
                            imgCols,
                            imgRows,
                            3);

    // Create a compatible ColorModel.
    ColorModel colorModel =
       PlanarImage.createColorModel(sampleModel);
    // Create a WritableRaster.
    Raster raster =
           RasterFactory.createWritableRaster(
             sampleModel,dbuffer,new Point(0,0));

    // Create a TiledImage using the SampleModel.
    TiledImage tiledImage = new TiledImage(0,0,
                             imgCols,imgRows,0,0,
                             sampleModel,
                             colorModel);

    // Set the data of the tiled image to be the
    // raster.
    tiledImage.setData(raster);

    // Save the image in a file using one of the
    // overloaded versions of the create method.
    // Note that other file types can be written
    // by using a different value for the third
    // parameter as in the following:
    //JAI.create("filestore",tiledImage,
    //                        "junk.tif","TIFF");
    JAI.create("filestore",tiledImage,
                             "junk.jpg","JPEG");
  }//end writeImageFile
}//end ImgMaker02 class
```

Listing 21

```
/*File ImgMod24.java.java
Copyright 2004, R.G.Baldwin

This program allows for multiple successive
convolutions using a fixed 3x3 flat convolution
filter.  The result approaches a Gaussian
filter as more successive convolutions are
performed.

This program normalizes the output so that the
largest color value in the output always matches
the largest color value in the input.  This may
or may not be desirable depending on the
circumstances.

This program is designed to be driven by the
```

program named ImgMod02. Enter the following at the command line to run this program.

```
java ImgMod02 ImgMod24 ImageFileName
```

This program illustrates the use of area (two-dimensional) convolution filtering to smooth or blur an image. In particular, it illustrates the process of multiple successive convolutions, which causes the convolution operation to approach the convolution of the image with a Gaussian filter.

The program displays two frames on the screen. The large frame on the left shows the original image at the top and the filtered image at the bottom. It also has a button labeled Replot at the very bottom.

The small frame on the right contains a TextField for user input. When the program starts running, this TextField displays the value 1. The value in the text field specifies the number of successive convolutions that are to be performed on the image using a flat 3x3 convolution filter.

To specify the number of convolutions, type an integer value into the TextField and click the Replot button. This will cause the process to start over and cause the filter to be applied the specified number of times before the new results are displayed.

This is a low-pass filter that suppresses high frequency changes in color values. It adds all of the pixel values for each color within the area covered by the filter. Then it moves to the next registration point and adds the pixel values then contained in the area. When the convolution operation is complete, all of the color values in the output are scaled so that the peak color value in the output matches the peak color value in the input.

Each pixel, except those in the outer edges of the image, is used as a registration point. The pixels around the outer edges are not used as registration points because that would cause the area to extend outside the valid pixel values.

The visual effect of applying this filter is to cause the image to go increasingly out of focus as the number of convolutions is increased. The effect is most obvious with images that have well-defined lines such as characters.

```
The transparency or alpha value of each pixel is
preserved.  If you don't see what you expect to
see when you run this program with a particular
image, it may be because your image contains
transparent areas.  This will be evidenced by the
yellow background color of the canvas showing
through the image.

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

class ImgMod24 extends Frame implements
                                       ImgIntfc02{

  int numberConvolutions;
  String inputData;//Obtained via the TextField
  TextField input;//User input field

  ImgMod24(){//constructor
    setLayout(new FlowLayout());

    Label instructions = new Label(
                "Number of convolutions/replot.");
    add(instructions);

    input = new TextField("1",5);
    add(input);

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

  //This method is required by ImgIntfc02.  This
  // method applies the convolution filter
  // to the incoming 3D array of pixel data and
  // returns a normalized filtered 3D array of
  // pixel data.  The output array is normalized
  // such that the peak output color value
  // matches the peak input color value.
  public int[][][] processImg(
                            int[][][] threeDPix,
                            int imgRows,
                            int imgCols){

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

    //Get numberConvolutions value from the
    // TextField
    numberConvolutions = Integer.parseInt(
                                 input.getText());
```

```
    //Make a working copy of the 3D array to
    // avoid making permanent changes to the
    // original image data.  Get and save the
    // maximum value along the way.
    int inputPeak = 0;
    int colorValue = 0;
    int[][][] working3D =
                    new int[imgRows][imgCols][4];
    for(int row = 0;row < imgRows;row++){
      for(int col = 0;col < imgCols;col++){
        working3D[row][col][0] =
                          threeDPix[row][col][0];
        colorValue = threeDPix[row][col][1];
        working3D[row][col][1] = colorValue;
        if(colorValue > inputPeak){
          inputPeak = colorValue;
        }//end if

        colorValue = threeDPix[row][col][2];
        working3D[row][col][2] = colorValue;
        if(colorValue > inputPeak){
          inputPeak = colorValue;
        }//end if

        colorValue = threeDPix[row][col][3];
        working3D[row][col][3] = colorValue;
        if(colorValue > inputPeak){
          inputPeak = colorValue;
        }//end if

      }//end inner loop
    }//end outer loop
    System.out.println(
                    "inputPeak = " + inputPeak);

    //Create an empty output array of the same
    // size as the incoming array.
    int[][][] output =
                    new int[imgRows][imgCols][4];

    //Copy all alpha values from input to output.
    for(int row = 0;row < imgRows;row++){
      for(int col = 0;col < imgCols;col++){
        output[row][col][0] =
                          working3D[row][col][0];
      }//end inner loop
    }//end outer loop

    //Perform the convolution one or more times
    // in succession
    for(int cnt = 0;
                cnt < numberConvolutions;cnt++){

      //Use nested for loops to treat each pixel
      // (other than those along the edges of the
      // image) as registration points and to
```

```
      // perform the two-dimensional convolution
      // using a shift-sum-scale approach. Note
      // that this algorithm is somewhat
      // different and probably more efficient
      // than the algorithm used in the program
      // named ImgMod12.  However, it is also
      // less flexible in terms of the shapes
      // of the convolution filters that can be
      // used.
      try{
        //Iterate on each pixel as a registration
        // point.
        for(int row = 0 + 1;row < imgRows - 2;
                                          row++){
          for(int col = 0 + 1;
                       col < imgCols - 2;col++){

            int redSum =
                 working3D[row - 1][col - 1][1] +
                 working3D[row - 1][col - 0][1] +
                 working3D[row - 1][col + 1][1] +
                 working3D[row - 0][col - 1][1] +
                 working3D[row - 0][col - 0][1] +
                 working3D[row - 0][col + 1][1] +
                 working3D[row + 1][col - 1][1] +
                 working3D[row + 1][col - 0][1] +
                 working3D[row + 1][col + 1][1];

            int greenSum =
                 working3D[row - 1][col - 1][2] +
                 working3D[row - 1][col - 0][2] +
                 working3D[row - 1][col + 1][2] +
                 working3D[row - 0][col - 1][2] +
                 working3D[row - 0][col - 0][2] +
                 working3D[row - 0][col + 1][2] +
                 working3D[row + 1][col - 1][2] +
                 working3D[row + 1][col - 0][2] +
                 working3D[row + 1][col + 1][2];


            int blueSum =
                 working3D[row - 1][col - 1][3] +
                 working3D[row - 1][col - 0][3] +
                 working3D[row - 1][col + 1][3] +
                 working3D[row - 0][col - 1][3] +
                 working3D[row - 0][col - 0][3] +
                 working3D[row - 0][col + 1][3] +
                 working3D[row + 1][col - 1][3] +
                 working3D[row + 1][col - 0][3] +
                 working3D[row + 1][col + 1][3];

            //Store the convolution output values
            // in the output array.
            output[row][col][1] = redSum;
            output[row][col][2] = greenSum;
            output[row][col][3] = blueSum;
```

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

    }catch(Exception e){
      e.printStackTrace();
    }//end catch

    //Normalize output peak value to match
    // input peak value.
    //First get output peak value
    int outputPeak = 0;
    for(int row = 0;row < imgRows;row++){
      for(int col = 0;col < imgCols;col++){
        if(output[row][col][1] > outputPeak){
          outputPeak = output[row][col][1];
        }//end if
        if(output[row][col][2] > outputPeak){
          outputPeak = output[row][col][2];
        }//end if
        if(output[row][col][3] > outputPeak){
          outputPeak = output[row][col][3];
        }//end if
      }//end inner loop
    }//end outer loop
    //System.out.println(
    //           "outputPeak = " + outputPeak);

    //Normalize to peak value
    double outputScale =
                ((double)inputPeak)/outputPeak;
    for(int row = 0;row < imgRows;row++){
      for(int col = 0;col < imgCols;col++){
        output[row][col][1] =
           (int)(output[row][col][1]*
                                   outputScale);
        output[row][col][2] =
           (int)(output[row][col][2]*
                                   outputScale);
        output[row][col][3] =
           (int)(output[row][col][3]*
                                   outputScale);
      }//end inner loop
    }//end outer loop

    //Copy output into input to prepare for
    // another convolution (no need to copy
    //alpha)
    for(int row = 0;row < imgRows;row++){
      for(int col = 0;col < imgCols;col++){
        working3D[row][col][1] =
                           output[row][col][1];
        working3D[row][col][2] =
                           output[row][col][2];
        working3D[row][col][3] =
                           output[row][col][3];
```

```
        }//end inner loop
      }//end outer loop
    }//end for loop on numberConvolutions

    System.out.println("Processing Done");
    //Return a reference to the array containing
    // the filtered pixels.
    return output;

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

}//end class ImgMod24
```

Listing 22

```
/*File ImgMod12.java
Copyright 2004, R.G.Baldwin

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

java ImgMod02 ImgMod12 ImageFileName

This program illustrates the use of area
(two-dimensional) convolution filtering to blur
an image.

The program displays two frames on the screen.
The large frame on the left shows the original
image at the top and the filtered image at the
bottom.  It also has a button labeled Replot at
the very bottom.

The small frame on the right contains a TextField
for user input.  When the program starts running,
this TextField displays the size of the default
convolution area in pixels.

To modify the convolution area, type a number
into the TextField and click the Replot button.
The new filter will be applied to the image and
the filtered image will be displayed.

The program supports non-square convolution area
values of 1, 2, 3, 4, 6, and 8 pixels.  (The
shape of the convolution area is shown as a grid
of X characters on the screen.)

Area values of 0, 5, and 7 are not supported.

In addition, the program supports all area values
that are perfect squares beginning with an area
value of 4 pixels.  However, for area values
```

```
greater than 9, the  value entered by the user is
automatically rounded to the nearest perfect
square before processing takes place.  For
example, if the user enters 10, the actual area
used for convolution will be a square with 3
pixels on each side.  If the user enters 15, the
area used for convolution will be a square with 4
pixels on each side.

The convolution operator is a box with each
coefficient having a value of 1.  (See
discussion of normalization later.)

This is a low-pass filter that suppresses high
frequency changes in color values.  It adds
all of the pixel values for each color within the
area covered by the filter.  Then it moves to the
next registration point and adds the pixel
values then contained in the area.

Every pixel, except those in the outer edges of
the image is used as a registration point.  The
pixels around the outer edges are not used as
registration points because that would cause the
area to extend outside the valid pixel values.

Once the convolution process is finished, the
output data is normalized such that the peak
color value in the output matches the peak color
value in the input.  This may, or may not be
appropriate depending on the circumstances.
However, it does preserve the dynamic range of
the display.

The visual effect of applying this filter is to
cause the image to go increasingly out of focus
as the size of the area is increased.  The effect
is most obvious with images that have well
defined lines such as characters.

The transparency or alpha value of each pixel is
preserved.  If you don't see what you expect to
see when you run this program with a particular
image, it may be because your image contains
transparent areas.  This will be evidenced by the
yellow background color of the canvas showing
through the image.

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

class ImgMod12 extends Frame implements
                                      ImgIntfc02{

  int area;//The area value in pixels
```

```
String inputData;//Obtained via the TextField
TextField input;//User input field

ImgMod12(){//constructor
  setLayout(new FlowLayout());

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

  input = new TextField("2",5);
  add(input);

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

//This method is required by ImgIntfc02.  This
// method applies the convolution filter
// to the incoming 3D array of pixel data and
// returns a filtered 3D array of pixel data.
public int[][][] processImg(
                          int[][][] threeDPix,
                          int imgRows,
                          int imgCols){

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

  //Get area value from the TextField
  area = Integer.parseInt(input.getText());

  //Create an empty output array of the same
  // size as the incoming array.
  int[][][] output =
                  new int[imgRows][imgCols][4];

  //Make a working copy of the 3D array to
  // avoid making permanent changes to the
  // original image data.  Get and save the
  // maximum value along the way.
  int inputPeak = 0;
  int colorValue = 0;
  int[][][] working3D =
                  new int[imgRows][imgCols][4];
  for(int row = 0;row < imgRows;row++){
    for(int col = 0;col < imgCols;col++){
      working3D[row][col][0] =
                        threeDPix[row][col][0];
      colorValue = threeDPix[row][col][1];
      working3D[row][col][1] = colorValue;
      if(colorValue > inputPeak){
        inputPeak = colorValue;
      }//end if
```

```
      colorValue = threeDPix[row][col][2];
      working3D[row][col][2] = colorValue;
      if(colorValue > inputPeak){
        inputPeak = colorValue;
      }//end if

      colorValue = threeDPix[row][col][3];
      working3D[row][col][3] = colorValue;
      if(colorValue > inputPeak){
        inputPeak = colorValue;
      }//end if

    }//end inner loop
  }//end outer loop
  System.out.println(
                  "inputPeak = " + inputPeak);

  //Copy all alpha values from input to output.
  for(int row = 0;row < imgRows;row++){
    for(int col = 0;col < imgCols;col++){
      output[row][col][0] =
                        working3D[row][col][0];
    }//end inner loop
  }//end outer loop

  //The following three variables are used to
  // accumulate the products of the pixel color
  // values and the convolution filter
  // coefficients.
  int redSum = 0;
  int greenSum = 0;
  int blueSum = 0;

  //The following variables are used for
  // control purposes while performing the
  // sum of products operation using for loops.
  int rowNo = 0;
  int colNo = 0;
  int row = 0;
  int col = 0;
  int firstRow = 0;
  int lastRow = 0;
  int firstCol = 0;
  int lastCol = 0;
  int minusRow = 0;
  int plusRow = 0;
  int minusCol = 0;
  int plusCol = 0;

  //The following switch statement is used to
  // set the control variables listed above for
  // area values of 1, 2, 3, 4, 6, and 8 on
  // an individual area basis.
  //Area values of 5 and 7 are not supported.
  //Area values of 9 and greater default to
```

```
    // the nearest perfect square, such as 9, 16,
    // 25, 36, etc.
    switch(area){
      case 0:
        System.out.println(
                    "Area value 0 not supported");
        break;
      case 1://A single pixel reproduces image
        firstRow = 0;
        lastRow = imgRows;
        firstCol = 0;
        lastCol = imgCols;
        minusRow = 0;
        plusRow = 0;
        minusCol = 0;
        plusCol = 0;
      break;
      case 2://Two pixels in a row
        firstRow = 0;
        lastRow = imgRows;
        firstCol = 1;
        lastCol = imgCols;
        minusRow = 0;
        plusRow = 0;
        minusCol = 1;
        plusCol = 0;
      break;
      case 3://Three pixels in a row
        firstRow = 0;
        lastRow = imgRows;
        firstCol = 1;
        lastCol = imgCols - 1;
        minusRow = 0;
        plusRow = 0;
        minusCol = 1;
        plusCol = 1;
      break;
      case 4://Four pixels in a square
        firstRow = 1;
        lastRow = imgRows;
        firstCol = 1;
        lastCol = imgCols;
        minusRow = 1;
        plusRow = 0;
        minusCol = 1;
        plusCol = 0;
      break;
      case 5:
        System.out.println(
                    "Area value 5 not supported");
      break;
      case 6://Two rows of 3 pixels
        firstRow = 1;
        lastRow = imgRows;
        firstCol = 1;
        lastCol = imgCols - 1;
```

```
        minusRow = 1;
        plusRow = 0;
        minusCol = 1;
        plusCol = 1;
      break;
      case 7:
        System.out.println(
                   "Area value 7 not supported");
      break;
      case 8://Two rows of 4
        firstRow = 1;
        lastRow = imgRows;
        firstCol = 2;
        lastCol = imgCols - 1;
        minusRow = 1;
        plusRow = 0;
        minusCol = 2;
        plusCol = 1;
      break;
      //Default to nearest perfect square for
      // area values greater than 8.
      default:
        //Get the side of the square area,
        // rounded to the nearest square.
        double dSide = Math.sqrt(area);
        int side = (int)Math.round(dSide);

        //Set the area value to the nearest
        // perfect square.  This is necessary
        // because it is used to scale the
        // accumulated values later.
        area = side*side;

        //Because a square area with an even
        // number of pixels on a side doesn't
        // have a pixel at the center, it must
        // be treated differently from a square
        // area with an odd number of pixels on a
        // side.  For the even case, the area
        // above and to the left of the
        // registration point is slightly greater
        // than the area below and to the right.
        if(side%2 == 0){//side is even
          firstRow = side/2;
          lastRow = imgRows - side/2 + 1;
          firstCol = side/2;
          lastCol = imgCols - side/2 + 1;
          minusRow = side/2;
          plusRow = side/2 - 1;
          minusCol = side/2;
          plusCol = side/2 -1;
        }else{//side is odd
          firstRow = side/2;
          lastRow = imgRows - side/2;
          firstCol = side/2;
          lastCol = imgCols - side/2;
```

```
        minusRow = side/2;
        plusRow = side/2;
        minusCol = side/2;
        plusCol = side/2;
      }//end else
    }//end switch statement

    //Use nested for loops to treat each pixel
    // (other than those along the edges of the
    // image) as registration points and to
    // perform the two-dimensional convolution.
    try{
      //First iterate on each pixel as a
      // registration point.
      for(row = firstRow;row < lastRow;row++){
        for(col = firstCol;col < lastCol;col++){

          //Now use the registration point as a
          // base and iterate on the pixels
          // contained within the area covered by
          // the convolution filter.  Display a
          // grid of X characters on the screen
          // showing the shape of the area
          // covered by the convolution filter.
          // Display the grid only once while
          // processing the first registration
          // point.
          for(rowNo = row - minusRow;
                rowNo <= row + plusRow;rowNo++){

            //Start a new line in the grid of X
            // characters.
            if((row == firstRow)
                          && (col == firstCol)){
              System.out.println();
            }//end if

            for(colNo = col - minusCol;
                colNo <= col + plusCol;colNo++){

              //Display the next X in the grid of
              // X characters.
              if((row == firstRow)
                          && (col == firstCol)){
                System.out.print("X");
              }//end if

              //Accumulate the pixel values
              // multiplied by the coefficient
              // values in the convolution
              // filter.  Note that all
              // coefficients have a value of 1.
              // The accumulated value will later
              // be divided by the area, causing
              // the effective values of the
              // coefficients to be the
```

```
            // reciprocal of the area.
            redSum +=
                    working3D[rowNo][colNo][1];
            greenSum +=
                    working3D[rowNo][colNo][2];
            blueSum +=
                    working3D[rowNo][colNo][3];
          }//end for loop on y
        }//end for loop on x

        //Store the accumlator values in the
        // output array.
        output[row][col][1] = redSum;
        output[row][col][2] = greenSum;
        output[row][col][3] = blueSum;

        //Clear the accumulators in preparation
        // for processing the next registration
        // point.
        redSum = 0;
        greenSum = 0;
        blueSum = 0;

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

  }catch(Exception e){
    e.printStackTrace();
  }//end catch

  //Normalize output peak value to match
  // input peak value.
  //First get output peak value
  int outputPeak = 0;
  for(row = 0;row < imgRows;row++){
    for(col = 0;col < imgCols;col++){
      if(output[row][col][1] > outputPeak){
        outputPeak = output[row][col][1];
      }//end if
      if(output[row][col][2] > outputPeak){
        outputPeak = output[row][col][2];
      }//end if
      if(output[row][col][3] > outputPeak){
        outputPeak = output[row][col][3];
      }//end if
    }//end inner loop
  }//end outer loop

  //Normalize to peak value
  double outputScale =
              ((double)inputPeak)/outputPeak;
  for(row = 0;row < imgRows;row++){
    for(col = 0;col < imgCols;col++){
      output[row][col][1] =
         (int)(output[row][col][1]*
                                  outputScale);
```

```
        output[row][col][2] =
           (int)(output[row][col][2]*
                                 outputScale);
        output[row][col][3] =
           (int)(output[row][col][3]*
                                 outputScale);
      }//end inner loop
    }//end outer loop

    //Return a reference to the array containing
    // the filtered pixels.
    return output;

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

}//end class ImgMod12
```

Listing 23

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

Keywords
Java pixel convolution filter Gaussian smooth blur image jpg color linear DSP 3D 2D

-end-