Processing Image Pixels, Creating Visible Watermarks in Java

Learn how to write a Java program that can be used to add five different types of visible watermarks to an image.

Published: December 19, 2006
By Richard G. Baldwin

Java Programming Notes # 418

Preface

Digital watermarking

Paraphrasing Wikipedia, **digital watermarking** is a technique that allows an individual to add copyright notices or other verification messages to digital audio, video, or image signals and documents. Digital watermarks may be visible, hidden, or a combination of the two.

Visible watermarks

This lesson will deal with visible watermarks, as shown in the bottom image of Figure 1.

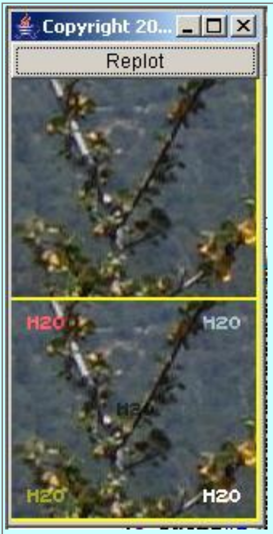

Figure 1 Visible Watermarks.

The bottom image in Figure 1 is a modified version of the original image shown at the top. The bottom image was modified to contain five different kinds of visible watermarks in the four corners and at the center of the image.

A program to add visible watermarks

In this lesson, I will present and explain a program named **ImgMod36**, which was used to produce the images shown in Figure 1. You will learn how to write a Java program that can be used to add five different types of visible watermarks to an image.

Hidden watermarks

A future lesson will deal with hidden watermarks similar to those shown in the center image of the right panel of Figure 2.

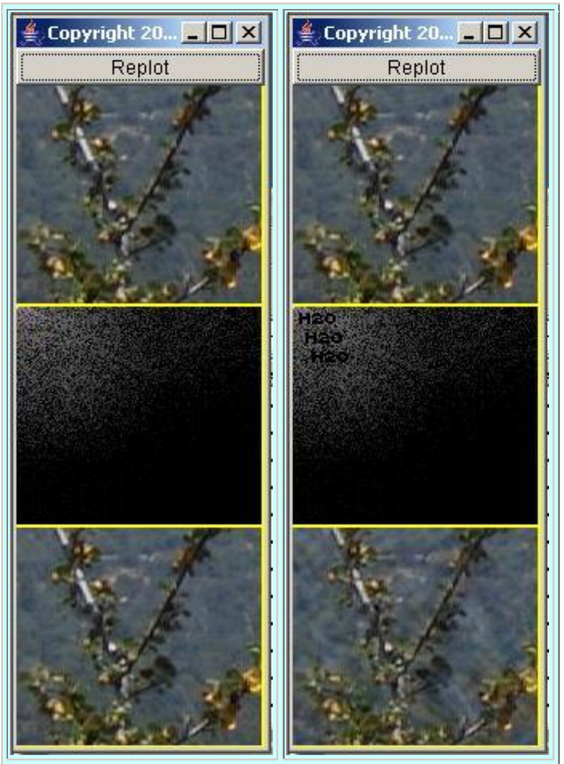

Figure 2 **Hidden Watermarks.**

Briefly, the images in the left panel of Figure 2 show an original image at the top and a replica of the original image at the bottom with no hidden watermarks having been added. The images in the right panel of Figure 2 also show the original image at the top and a replica of the original image at the bottom. However, in this case, three watermarks been hidden in the frequency-domain representation of the image shown by the middle image.

Viewing tip

You may find it useful to open another copy of this lesson in a separate browser window. That will make it easier for you to scroll back and forth among the different listings and figures while you are reading about them.

Supplementary material

I recommend that you also study the other lessons in my extensive collection of online Java tutorials. You will find those lessons published at Gamelan.com. However, as of the date of this writing, Gamelan doesn't maintain a consolidated index of my Java tutorial lessons, and sometimes they are difficult to locate there. You will find a consolidated index at www.DickBaldwin.com.

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

Preview

The program named ImgMod36

There are many different ways to add watermarks to an image. The purpose of this program is to illustrate the creation of five different types of visible watermarks as shown in the bottom image of Figure 1:

1. A high intensity watermark in a single color plane *(red, top left corner)*.
2. A watermark that is the same color as the original image pixels but with twice the intensity *(top right corner)*.
3. A watermark that is the same color as the original image pixels but with only half the intensity *(center)*.
4. A watermark for which the alpha *(transparency)* values of the pixels are half of the original values *(bottom left corner)*.
5. A high intensity white watermark *(bottom right corner)*.

Driven by ImgMod04a

This program is designed to be driven by the image processing framework program named **ImgMod04a**. The framework program was developed and explained in the earlier lesson entitled Processing Image Pixels, An Improved Image-Processing Framework in Java.

Operation

To run this program, enter the following at the command line.

```
java ImgMod04a ImgMod36 ImageFileName
```

where **ImageFileName** is the name of a .gif or .jpg file, including the extension.

A single display frame

The program displays a single frame on the screen as shown in Figure 1. The frame contains the original image at the top and a replica of the original image with five watermarks added at the bottom.

Replot button
The frame also contains a **Replot** button. However, because this program does not allow the user to enter parameters to modify the behavior of the program at runtime, clicking the **Replot** button has little or no effect.

An output JPEG file

Each time the program is run, or the **Replot** button is clicked, the bottom image containing the watermarks is written into a JPEG file named **junk.jpg**. For example, Figure 3 shows a screen capture of the contents of the output JPEG file produced for Figure 1.

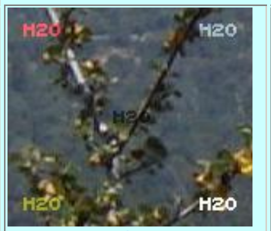

Figure 3 Output JPEG File.

If a file having name **junk.jpg** already exists in the current directory, it is overwritten.

The processImg method

In order for a class to be compatible with the framework program named **ImgMod04a**, the class must implement the interface named **ImgIntfc04a**. The interface named **ImgIntfc04a** declares a method named **processImg**. Therefore, the class named **ImgMod36** must define an image processing method named **processImg**.

The image processing method named **processImg** is executed by the program named **ImgMod04a** upon startup, and each time thereafter that the user clicks the **Replot** button at the top of Figure 1.

In this program, the method named **processImg** modifies the image pixels in five selected locations in the original image to add the watermarks described above and shown in Figure 1. Then it returns the modified image, which is displayed by the program named **ImgMod04a**.

Class files required

This program requires access to the following class files plus some inner classes that are defined inside the following classes:

- ImgIntfc04a.class
- ImgMod04a.class
- ImgMod36.class

The source code for the first two classes was provided in the earlier lesson entitled Processing Image Pixels, An Improved Image-Processing Framework in Java. The source code for the third class is provided in Listing 13.

Program testing

The program was tested using J2SE 5.0 and WinXP.

Discussion and Sample Code

Will explain in fragments

As is my normal practice, I will explain this program in fragments. You can view a complete listing of the class named **ImgMod36** in Listing 13 near the end of the lesson.

Basically, this class consists of the method named **processImg**, a method named **addWatermark**, and several utility methods that are used to perform common operations on the color planes that represent the image in a 3D array of type **double**.

The processImg method

The class named **ImgMod36** and the method named **processImg** begin in Listing 1.

```
class ImgMod36 implements ImgIntfc04a{
  public double[][][] processImg(double[][][]
threeDPix){

    int imgRows = threeDPix.length;
    int imgCols = threeDPix[0].length;

    //Make a working copy of the 3D pixel array
to avoid
    // making permanent changes to the original
image data.
    double[][][] workingCopy =
copy3DArray(threeDPix);

    //Declare a working plane.
    double[][] workingPlane;
```

Listing 1

Understanding the ImgMod04a class

It will be very helpful if you understand the image processing framework program named **ImgMod04a** before attempting to understand this program. You will find an explanation of **ImgMod04a** in the earlier lesson entitled Processing Image Pixels, An Improved Image-Processing Framework in Java.

Processing the image

Regardless of whether or not you already understand **ImgMod04a**, the method named **processImg** receives an image formatted into a 3D array of type **double** and returns a modified image in a 3D array of type **double**.

Four image planes

The 3D array describes four image planes consisting of an *alpha* or *transparency* plane, a *red* plane, a *green* plane, and a *blue* plane in that order. All of the necessary conversions between type *unsigned byte***,** which is the standard format for image pixel data, and type **double** are handled by the class named **ImgMod04a**. Thus, the method named **processImg** has only to contend with data of type **double**.

The code in Listing 1 begins by finding the dimensions in pixels of the image planes. Then it invokes the method named **copy3DArray** to make a working copy of the 3D pixel array to avoid making permanent changes to the original image data.

Straightforward code
The code in the method named **copy3DArray** is straightforward and shouldn't require an explanation. You can view the method in its entirety in Listing 13.

A working 2D plane

Finally, the code in Listing 1 declares a 2D array of type **double** that will be used as a working plane. This plane is used to avoid making individual working copies of each of the four image planes and reduces the memory requirements for processing large images.

Extract and process the alpha plane

The code in Listing 2:

- Invokes the **extractPlane** method to extract the alpha plane from the 3D working array into the 2D working plane.
- Invokes the **addWatermark** method to add watermarks to the working plane, passing the working plane and the identification of the alpha plane, 0, to the **addWatermark** method.
- Invokes the **insertPlane** method to insert the modified working plane back into the 3D working array, replacing the alpha plane previously stored there.

```
    //Extract and process the alpha plane.
    workingPlane = extractPlane(workingCopy,0);

    //Add the watermarks.
    addWatermark(workingPlane,0);

    //Insert the alpha plane back into the
working array.
    insertPlane(workingCopy,workingPlane,0);
```

Listing 2

The **extractPlane** method and the **insertPlane** method are straightforward and shouldn't require an explanation. You can view those methods in their entirety in Listing 13.

The addWatermark method

The **addWatermark** method is at the heart of this lesson. Therefore, I will set the discussion of the **processImg** method aside long enough to explain the **addWatermark** method.

The **addWatermark** method begins in Listing 3. The purpose of this method is to add visible watermarks to a 2D array of pixel data of type **double**. There are many ways to modify the pixel data to add watermarks. This method illustrates five of those ways.

```
  void addWatermark(double[][] plane,int 
color){
    int imgRows = plane.length;
    int imgCols = plane[0].length;

Listing 3
```

The incoming parameters

The **addWatermark** method receives two incoming parameters. One parameter is a reference to the 2D array of pixel data that is to be modified to add the watermarks.

The other parameter is an **int** value that identifies the plane according to the following key:

- 0 - alpha plane
- 1 - red plane
- 2 - green plane
- 3 - blue plane

The dimensions of the 2D image plane

The **addWatermark** code in Listing 3 determines the dimensions of the image plane in pixels.

Create the watermark

Listing 4 creates an array object containing the watermark. The watermark consists of the characters **H2O** described by **int** values of 1 and 0 arranged in a row-column format. Each **int** value in the array will be used later to determine whether or not to modify a pixel in the image. Thus, there is a one-to-one correspondence between the **int** values in the watermark array and the pixels in the image.

```
    int[][] watermark = new int[][]{
      {1,1,0,0,0,1,1, 0, 0,1,1,1,1,1,0, 0, 
0,0,1,1,1,0,0},
      {1,1,0,0,0,1,1, 0, 1,1,1,1,1,1,1, 0, 
0,1,1,1,1,1,0},
      {1,1,1,1,1,1,1, 0, 1,0,0,0,0,1,1, 0, 
1,1,0,0,0,1,1},
      {1,1,1,1,1,1,1, 0, 0,0,1,1,1,1,0, 0, 
1,1,0,0,0,1,1},
      {1,1,0,0,0,1,1, 0, 0,1,1,1,1,0,0, 0, 
1,1,0,0,0,1,1},
      {1,1,0,0,0,1,1, 0, 1,1,0,0,0,0,0, 0, 
1,1,0,0,0,1,1},
      {1,1,0,0,0,1,1, 0, 1,1,1,1,1,1,1, 0,
0,1,1,1,1,1,0},
      {1,1,0,0,0,1,1, 0, 1,1,1,1,1,1,1, 0,
0,0,1,1,1,0,0},
    };//end array

    int wmRows = watermark.length;
    int wmCols = watermark[0].length;
```

Listing 4

Listing 4 also gets and saves the dimensions of the watermark array.

Iterate on watermark rows and columns

Listing 5 shows the opening statements of a pair of nested **for** loops that iterate on the rows and columns of the watermark.

```
    for(int row = 0;row < wmRows;row++){
      for(int col = 0;col < wmCols;col++){
        if(watermark[row][col] == 1){//Ignore 0
values.
```

Listing 5

The pair of nested **for** loops that begin in Listing 5 encapsulate five sections of code that are used to modify pixel values in order to produce the five different types of watermarks shown in Figure 1.

Selecting the pixels to be modified

The **if** statement in Listing 5 causes those five sections of code to be executed for every row-column combination where the watermark value is 1 and causes the five sections of code to be skipped for every row-column combination where the watermark value is 0.

High-intensity watermark on the red plane

Listing 6 tests the incoming parameter value named **color** *(that specifies the plane)* and ignores all planes except the red plane that is identified by a value of 1. For that case, the current color value of the specified pixel is replaced by the maximum possible color value of 255.

```
          if(color == 1){//Modify red plane
only.
            plane[row+10][col+10] = 255.0;
          }//end if
```

Listing 6

Do the arithmetic
I will let you do the arithmetic

This watermark is placed in the upper-left corner of the bottom image as shown in Figure 1.

yourself to see how it ends up in the upper-left corner.

A watermark in the upper right corner

Listing 7 places a watermark in the upper right corner of the screen as shown in the bottom image of Figure 1. The color of the watermark is basically the same as the color of the image at that location but the intensity of each pixel in the watermark is twice the intensity of the original pixel.

```
          if(color != 0){//Don't modify the 
alpha plane.
            plane[row+10][imgCols-wmCols-
10+col] =
                  plane[row+10][imgCols-wmCols-
10+col]*2.0;
            plane[row+10][imgCols-wmCols-
10+col] =
               clamp(plane[row+10][imgCols-
wmCols-10+col]);
          }//end if
```

Listing 7

Need to clamp the values

This procedure can lead to pixels having color values that are greater than the maximum allowed value of 255. This is dealt with by invoking the method named **clamp** to guarantee that the value is within the range of 0 to 255 inclusive. The **clamp** method is straightforward and shouldn't require an explanation. It can be viewed in its entirety in Listing 13.

Intensity
Note that the intensity of a red, green, or blue pixel color can range from 0 to 255 with 0 being black and 255 being full intensity.

Doesn't modify the transparency

Note that this procedure doesn't change the pixels in the alpha plane identified by a **color** value of 0.

A watermark in the center of the image

Listing 8 places a watermark in the center of the image as shown by the dark watermark in the center of the bottom image in Figure 1. The color of the watermark is basically the same as the color of the image at that location but the intensity of each pixel in the watermark is half the intensity of the original pixel.

```
          if(color != 0){//Don't modify alpha 
plane.
            plane[imgRows-(imgRows/2+wmRows/2)+
row]
                       [imgCols-
(imgCols/2+wmCols/2)+col] =
                  plane[imgRows-
(imgRows/2+wmRows/2)+row]
                    [imgCols-
(imgCols/2+wmCols/2)+col]*0.5;
          }//end if

Listing 8
```

Note once again that the values of the pixels in the alpha plane are not modified.

A watermark that manipulates transparency values

The construction of the watermark in the lower left corner of Figure 1 is considerably different from the others. This watermark, which is constructed by Listing 9, doesn't change the values of the red, green, and blue pixels. Rather, it causes those pixels to be more transparent allowing the background color of the frame to show through.

```
          if(color == 0){//Modify alpha plane
only.
            plane[imgRows-wmRows-
10+row][col+10] =
                  plane[imgRows-wmRows-
10+row][col+10]/2.0;
          }//end if

Listing 9
```

The background color of the frame in Figure 1 is yellow. Therefore, this pixel appears to be yellow in Figure 1.

A white watermark

Listing 10 places a white watermark in the lower right corner of the image as shown in Figure 1. Listing 10 doesn't modify the transparency value.

```
          if(color != 0){//Don't modify the
alpha plane.
            plane[imgRows-wmRows-10+row]
                           [imgCols-wmCols-
10+col] = 255.0;
          }//end if

        }//end if on watermark pixel value
      }//end inner loop on wmCols
    }//end outer loop on wmRows

  }//end addWatermark
```

Listing 10

Listing 10 also signals the end of the method named **addWatermark**.

Process the remaining color planes

Returning now to the **processImg** method, the alpha plane has been processed. Listing 11 processes the red, green, blue color planes using exactly the same methodology as was used to process the alpha plane.

```
    //Extract and process the red color plane.
    workingPlane = extractPlane(workingCopy,1);
    addWatermark(workingPlane,1);
    insertPlane(workingCopy,workingPlane,1);

    //Extract and process the green color 
plane.
    workingPlane = extractPlane(workingCopy,2);
    addWatermark(workingPlane,2);
    insertPlane(workingCopy,workingPlane,2);

    //Extract and process the blue color plane.
    workingPlane = extractPlane(workingCopy,3);
    addWatermark(workingPlane,3);
    insertPlane(workingCopy,workingPlane,3);
```

Listing 11

Return the modified image array

The alpha plane and all three color planes have now been processed. The results are stored in the working copy of the original pixel array. Listing 12 returns the array containing the modified image to the calling method.

```
    return workingCopy;

  }//end processImg method
```

Listing 12

Listing 12 also completes the discussion of the program code.

Susceptibility to modifications

As you will learn in future lessons that deal with hidden watermarks, one of the big issues involving watermarks has to do with the susceptibility of the watermark to modification of the image. For example, if it is easy to cause a watermark to disappear by modifying the image, the watermark may not be very useful in allowing an individual to add copyright notices or other verification messages to digital images.

Example of a modified watermarked image

Figure 4 shows the result of rotating, flipping, and scaling the image stored in the JPEG output file from Figure 1.

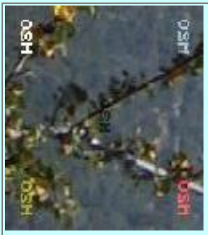

Figure 4 Modified JPEG File.

As you can see from Figure 4, visible watermarks of the types employed by this program tend to be very persistent, at least with respect to the kinds of modifications that were applied to this image.

Not necessarily true for hidden watermarks

However, as you will learn in future lessons that discuss hidden watermarks, achieving such persistence is much more difficult for hidden watermarks. In fact, sometimes the simple act of storing a watermarked image in a JPEG file can cause the hidden watermark to be lost. Other operations such as scaling the image can be even more detrimental to hidden watermarks.

Run the Program

I encourage you to copy the code from Listing 13 into your text editor, compile it, and execute it. Experiment with it, making changes, and observing the results of your changes. See if you can come up with some different and interesting ways to create visible watermarks by manipulating the alpha value and the three color values of the pixels.

Summary

In this lesson, I presented and explained a program that can be used to add five different types of visible watermarks to an image.

What's Next?

Future lessons will explain how to add hidden watermarks to an image. As you will see in those lessons, that task is somewhat more difficult than adding visible watermarks.

References

400 Processing Image Pixels using Java, Getting Started
402 Processing Image Pixels using Java, Creating a Spotlight
404 Processing Image Pixels Using Java: Controlling Contrast and Brightness
406 Processing Image Pixels, Color Intensity, Color Filtering, and Color Inversion
408 Processing Image Pixels, Performing Convolution on Images
410 Processing Image Pixels, Understanding Image Convolution in Java
412 Processing Image Pixels, Applying Image Convolution in Java, Part 1
414 Processing Image Pixels, Applying Image Convolution in Java, Part 2
416 Processing Image Pixels, An Improved Image-Processing Framework in Java

Complete Program Listing

A complete listing of the program discussed in this lesson is shown in Listing 13 below.

```
/*File ImgMod36.java
Copyright 2006, R.G.Baldwin

There are many different ways to add watermarks to an
image.  The purpose of this program is to illustrate the
creation of five different types of visible watermarks:

1. A high intensity watermark in a single color
   plane (red).
2. A watermark that is the same color as the original image
   pixels but twice as intense.
3. A watermark that is the same color as the original image
   pixels but with only half the intensity.
4. A watermark for which the alpha (transparency) value of
   the pixels is half of the original values.
5. A high intensity white watermark.

This program is designed to be driven by the image
processing framework named ImgMod04a.  To run this
program, enter the following at the command line.

java ImgMod04a ImgMod36 ImageFileName

where ImageFileName is the name of a .gif or .jpg file,
including the extension.

The program displays a single frame on the screen.  The
frame contains the original image at the top and a replica
of the original image with the watermarks added at the
bottom.  The frame also contains a Replot button.
However, because the program does not allow the user to
enter parameters to modify the behavior of the program at
runtime, clicking the Replot button has little or no
beneficial effect.
```

```
Each time that the program is run, or the Replot button
is clicked, the final image containing the watermarks is
written into a JPEG file named junk.jpg.  If a file
having that name already exists in the current directory,
it is overwritten.

This program contains an image processing method named
processImg, which is executed by the program named
ImgMod04a.  The method named processImg modifies the image
pixels in five selected locations to add the watermarks
described above.  Then it returns the modified image, which
is displayed by the program named ImgMod04a.

This program requires access to the following class files
plus some inner classes that are defined inside the
following classes:

ImgIntfc04a.class
ImgMod04a.class
ImgMod36.class

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

class ImgMod36 implements ImgIntfc04a{

  //This method is required by ImgIntfc04a.  It is called
  // at the beginning of the run and each time thereafter
  // that the user clicks the Replot button on the Frame
  // containing the images.  However, because this program
  // doesn't provide for user input, pressing the Replot
  // button is of no value.  It just displays the same
  // images again.
  public double[][][] processImg(double[][][] threeDPix){

    int imgRows = threeDPix.length;
    int imgCols = threeDPix[0].length;

    //Make a working copy of the 3D pixel array to avoid
    // making permanent changes to the original image data.
    double[][][] workingCopy = copy3DArray(threeDPix);

    //Declare a working plane.
    double[][] workingPlane;

    //Extract and process the alpha plane.
    workingPlane = extractPlane(workingCopy,0);
    addWatermark(workingPlane,0);
    //Insert the alpha plane back into the working array.
    insertPlane(workingCopy,workingPlane,0);

    //Extract and process the red color plane.
    workingPlane = extractPlane(workingCopy,1);
    addWatermark(workingPlane,1);
    insertPlane(workingCopy,workingPlane,1);
    //Extract and process the green color plane.
    workingPlane = extractPlane(workingCopy,2);
    addWatermark(workingPlane,2);
    insertPlane(workingCopy,workingPlane,2);

    //Extract and process the blue color plane.
    workingPlane = extractPlane(workingCopy,3);
    addWatermark(workingPlane,3);
    insertPlane(workingCopy,workingPlane,3);

    //The alpha plane and all three color planes have now
    // been processed.  The results are stored in the
    // working copy of the original pixel array.
    return workingCopy;

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

  //The purpose of this method is to extract a color plane
  // from the double version of an image and to return it
  // as a 2D array of type double.
  public double[][] extractPlane(
                            double[][][] threeDPixDouble,
                            int plane){

    int numImgRows = threeDPixDouble.length;
    int numImgCols = threeDPixDouble[0].length;

    //Create an empty output array of the same
    // size as a single plane in the incoming array of
    // pixels.
    double[][] output =new double[numImgRows][numImgCols];

    //Copy the values from the specified plane to the
    // double array.
    for(int row = 0;row < numImgRows;row++){
      for(int col = 0;col < numImgCols;col++){
        output[row][col] =
                          threeDPixDouble[row][col][plane];
      }//end loop on col
    }//end loop on row
    return output;
  }//end extractPlane
  //-----------------------------------------------------//

  //The purpose of this method is to insert a double 2D
  // plane into the double 3D array that represents an
  // image.
  public void insertPlane(double[][][] threeDPixDouble,
                          double[][] colorPlane,
                          int plane){

    int numImgRows = threeDPixDouble.length;
    int numImgCols = threeDPixDouble[0].length;

    //Copy the values from the incoming color plane to the
    // specified plane in the 3D array.
    for(int row = 0;row < numImgRows;row++){
      for(int col = 0;col < numImgCols;col++){
        threeDPixDouble[row][col][plane] =
                                              colorPlane[row][col];
      }//end loop on col
    }//end loop on row
  }//end insertPlane
  //-----------------------------------------------------//

  //This method copies a double version of a 3D pixel array
  // to an new pixel array of type double.
  double[][][] copy3DArray(double[][][] threeDPix){
    int imgRows = threeDPix.length;
    int imgCols = threeDPix[0].length;

    double[][][] new3D = new double[imgRows][imgCols][4];
    for(int row = 0;row < imgRows;row++){
      for(int col = 0;col < imgCols;col++){
        new3D[row][col][0] = threeDPix[row][col][0];
        new3D[row][col][1] = threeDPix[row][col][1];
        new3D[row][col][2] = threeDPix[row][col][2];
        new3D[row][col][3] = threeDPix[row][col][3];
      }//end inner loop
    }//end outer loop
    return new3D;
  }//end copy3DArray
  //-----------------------------------------------------//

  //The purpose of this method is to add watermarks to
  // a 2D array of pixel data. There are many ways to
  // modify the pixel data to add watermarks.  This
  // method illustrates five of those ways.
  void addWatermark(double[][] plane,int color){
    int imgRows = plane.length;
    int imgCols = plane[0].length;

    //Create an array containing the basic watermark. The
    // watermark consists of the characters H2O described
    // by values of 1 and 0.
    int[][] watermark = new int[][]{
      {1,1,0,0,0,1,1, 0, 0,1,1,1,1,1,0, 0, 0,0,1,1,1,0,0},
      {1,1,0,0,0,1,1, 0, 1,1,1,1,1,1,1, 0, 0,1,1,1,1,1,0},
      {1,1,1,1,1,1,1, 0, 1,0,0,0,0,1,1, 0, 1,1,0,0,0,1,1},
      {1,1,1,1,1,1,1, 0, 0,0,1,1,1,1,0, 0, 1,1,0,0,0,1,1},
      {1,1,0,0,0,1,1, 0, 0,1,1,1,1,0,0, 0, 1,1,0,0,0,1,1},
      {1,1,0,0,0,1,1, 0, 1,1,0,0,0,0,0, 0, 1,1,0,0,0,1,1},
      {1,1,0,0,0,1,1, 0, 1,1,1,1,1,1,1, 0, 0,1,1,1,1,1,0},
      {1,1,0,0,0,1,1, 0, 1,1,1,1,1,1,1, 0, 0,0,1,1,1,0,0},
    };//end array

    int wmRows = watermark.length;
    int wmCols = watermark[0].length;

    for(int row = 0;row < wmRows;row++){
      for(int col = 0;col < wmCols;col++){
        if(watermark[row][col] == 1){//Ignore 0 values.

          //Place a high intensity watermark only in the
          // red color plane of the image.  Place it in the
          // upper left.
          if(color == 1){//Modify red plane only.
            plane[row+10][col+10] = 255.0;
          }//end if

          //Place a watermark in the upper right area.
          // Make the color of the watermark be the
          // same as the color of the image but twice as
          // intense.
          if(color != 0){//Don't modify the alpha plane.
            plane[row+10][imgCols-wmCols-10+col] =
                  plane[row+10][imgCols-wmCols-10+col]*2.0;
            plane[row+10][imgCols-wmCols-10+col] =
               clamp(plane[row+10][imgCols-wmCols-10+col]);
          }//end if

          //Place a watermark in the center of the image.
          // Make the intensity of each color to be half of
          // the original intensity.
          if(color != 0){//Don't modify alpha plane.
            plane[imgRows-(imgRows/2+wmRows/2)+ row]
                       [imgCols-(imgCols/2+wmCols/2)+col] =
                  plane[imgRows-(imgRows/2+wmRows/2)+row]
                    [imgCols-(imgCols/2+wmCols/2)+col]*0.5;
          }//end if

          //Place a watermark in the lower left.  Make the
          // transparency value of each pixel to be half of
          // its original value.
          if(color == 0){//Modify alpha plane only.
            plane[imgRows-wmRows-10+row][col+10] =
                  plane[imgRows-wmRows-10+row][col+10]/2.0;
          }//end if


          //Place a high intensity white watermark in the
          // bottom-right.
          if(color != 0){//Don't modify the alpha plane.
            plane[imgRows-wmRows-10+row]
                           [imgCols-wmCols-10+col] = 255.0;
          }//end if

        }//end if on watermark pixel value
      }//end inner loop on wmCols
    }//end outer loop on wmRows

  }//end addWatermark
  //-----------------------------------------------------//

  //The purpose of this method is to clamp the incoming
  // value to guarantee that it falls in the range from 0
  // to 255 inclusive.
  double clamp(double data){
    if(data > 255.0){
      return 255.0;
    }else if(data < 0.0){
      return 0.0;
    }else{
      return data;
    }//end else
  }//end clamp
  //----------------------------------------------------//
}//end class ImgMod36
```

Listing 13

About the author

Richard Baldwin *is a college professor (at Austin Community College in Austin, TX) and private consultant whose primary focus is a combination of Java, C#, and XML. In addition to the many platform and/or language independent benefits of Java and C# applications, he believes that a combination of Java, C#, and XML will become the primary driving force in the delivery of structured information on the Web.*

Richard has participated in numerous consulting projects and he frequently provides onsite training at the high-tech companies located in and around Austin, Texas. He is the author of Baldwin's Programming Tutorials, which have gained a worldwide following among experienced and aspiring programmers. He has also published articles in JavaPro magazine.

In addition to his programming expertise, Richard has many years of practical experience in Digital Signal Processing (DSP). His first job after he earned his Bachelor's degree was doing DSP in the Seismic Research Department of Texas Instruments. (TI is still a world leader in DSP.) In the following years, he applied his programming and DSP expertise to other interesting areas including sonar and underwater acoustics.

Richard holds an MSEE degree from Southern Methodist University and has many years of experience in the application of computer technology to real-world problems.

Baldwin@DickBaldwin.com

Keywords
java image watermark pixel

-end-