Plotting 3D Surfaces using Java

Learn how to write a Java class that uses color to plot 3D surfaces in six different formats and a wide range of sizes. The class is extremely easy to use. You can incorporate the 3D plotting capability into your own programs by inserting a single statement into your programs.

Published: May 31, 2005
By Richard G. Baldwin

Java Programming, Notes # 1489

Preface

In one of my earlier lessons entitled Plotting Engineering and Scientific Data using Java, I published a generalized 2D plotting program, which makes it easy to cause other programs to display their outputs in 2D Cartesian coordinates. I have used that plotting program in numerous lessons since I published it. Hopefully, some of my readers have been using it as well.

In this lesson, I will present and explain a 3D surface plotting program that is also very easy to use.

Numerous Java graphics libraries are available from various locations on the web. Some are of high quality, and some are not. Unfortunately, many of those libraries have a rather substantial learning curve.

The purpose of the class

It is not the purpose of the class that I will provide to compete with those graphics libraries. Rather, this class is intended to make it possible for an experienced Java programmer to incorporate 3D surface plotting capability into a Java program with a learning curve of three minutes or less.

> *(If you are an experienced Java programmer, you can start your three-minute learning-curve clock right now.)*

How do you use the class?

All that's necessary to use this class to plot your own 3D surfaces is to copy and compile the source code in Listing 29 near the end of this lesson, and then include a statement similar to the following in your program:

```
new ImgMod29(data,blockSize,true,0);
```

The 3D surface data to be plotted

The parameter named **data** in the above statement is a reference to a 2D array of type **double** that contains the sampled elevation values of the surface to be plotted.

The granularity of the plot

The second parameter named **blockSize** specifies the size of one side of a square array of colored pixels in the final plot that will represent each elevation point on your 3D surface. Set this to 0 if you are unsure as to what size square you need.

> *(See the small white square at the center of the middle image in Figure 1. This is a nine-pixel square produced by a **blockSize** value of 3.)*

The optional axes

The third parameter specifies whether or not you want to have optional axes drawn on the plot. *(See Figure 3 for examples of plots with and without the axes.)* A parameter value of **true** causes the axes to be drawn. A value of **false** causes the axes to be omitted.

The plotting format

The fourth parameter is an integer that specifies the plotting format as follows:

- 0 - Grayscale *(linear, top-left image in Figure 2)*
- 1 - Color Shift *(linear, top-center image in Figure 2)*
- 2 - Color Contour *(linear, top-right image in Figure 2)*
- 3 - Grayscale with logarithmic data conversion *(bottom-left image in Figure 2)*
- 4 - Color Shift with logarithmic data conversion *(bottom-center image in Figure 2)*
- 5 - Color Contour with logarithmic data conversion *(bottom-right image in Figure 2)*

Just start with a value of 2 or 3 if you are unsure as to which format would be best for your application. Then try all six formats to see which one works best for you.

Extremely simple to use

The class couldn't be simpler to use.

(Your three-minute learning curve has expired. You now know how to use the class to incorporate 3D surface plotting in your Java programs.)

Will use in subsequent lessons

This 3D plotting class will be used in numerous future lessons involving such complex topics as the use of the 2D Fourier Transform to process images and the embedding of secret watermarks in images.

If you arrived at this page seeking a free Java program for plotting your 3D surfaces, you are in luck. Just copy the source code for the class in Listing 29 and feel free to use it as described above.

On the other hand, if you would like to learn how the class does what it does, and perhaps use your programming skills to improve it, keep reading. Hopefully, once you have finished the lesson, you will have learned quite a lot about plotting 3D surfaces using color in Java.

Viewing tip

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

Supplementary material

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

General Discussion

Displaying 3D data can be fairly difficult

One of the more difficult aspects of engineering and scientific computing is displaying three-dimensional *(3D)* surfaces in ways that are meaningful to persons who need to view and to analyze those surfaces. The basic problem is that it is necessary to display the 3D surface on a 2D media, such as a computer screen. Therefore, some compromise is always required.

Different approaches are available

Various approaches have been devised for accomplishing this objective including:

- Labeled Numeric Contour Plots
- Isometric Drawings

- Grayscale Plots
- Color Shift Plots
- Color Contour Plots
- Using light and shadows to render the surface in ways that simulate a photograph

Will provide and explain a program ...

I will provide and explain a program in this lesson that makes it very easy to display a 3D surface as a Grayscale Plot, a Color Shift Plot, or a Color Contour Plot. The program supports six different plotting formats. Three of those plotting formats are illustrated in Figure 1.

> *(The images shown in Figure 1 are different views of the wave-number spectrum resulting from performing a 2D Fourier Transform on a box in the space domain. The use of 2D Fourier Transforms will be the main topic of a future lesson.)*

Sample program output

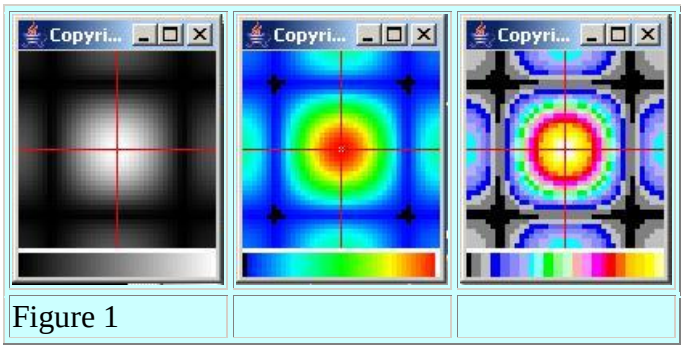

Figure 1

Figure 1 shows the same 3D surface plotted using three different plotting formats. Going from left to right, Figure 1 shows:

- Grayscale Plot
- Color Shift Plot
- Color Contour Plot

Grayscale Plot

The plot on the left in Figure 1 is the old standby method in which the elevation of each point on the surface is represented by a shade of gray with the highest elevation being white and the lowest elevation being black.

A calibration scale

Each of the images in Figure 1 shows a calibration scale immediately below the image of the surface. The calibration scale shows the color or shade of gray used to represent each elevation

on the surface with the color or shade of the lowest elevation on the left of the scale and the color or shade of the highest elevation on the right.

For example, the scale on the Grayscale image shows a smooth gradient from black to white going from left to right. The shade of gray shown at the midpoint on the calibration scale represents the elevation that is halfway between the lowest elevation and the highest elevation.

Color Shift Plot

The image in the center in Figure 1 shows the same surface plotted using a smooth gradient from blue at the low end through aqua, green, and yellow to red at the high end. In addition, this plotting format sets the lowest elevation to black and the highest elevation to white so that these two elevations are obvious in the plot.

> *(The highest elevation is indicated by the small white square at the center, and the lowest elevations are indicated by the black areas near the corners.)*

More information is conveyed

This plotting format can convey a great deal more information to a human observer than the Grayscale plotting format. This is because the human eye can discern more different colors than it can discern different shades of gray.

> *(Many years ago, when I was in the SONAR business, the general rule of thumb was that a typical human can discern only about seven shades of gray from black to white inclusive. Obviously a typical human can discern more than seven different colors.)*

The lowest elevations are obvious in the Color Shift Plot

By using black to indicate the lowest elevation, *(in addition to using shades of blue to indicate low elevations)*, it is easy to determine the exact locations of the lowest elevations in the Color Shift Plot in the center of Figure 1. On the other hand, the locations of the lowest elevations are not discernable in the Grayscale Plot at the left Figure 1.

Four minor peaks are obvious in the Color Shift Plot

Also, it is obvious from the Color Shift Plot that the surface has four minor peaks at the edges of the plot. Although the Grayscale Plot has a slight hint of those minor peaks, they are certainly not obvious.

By comparing the color at the top of the minor peaks to the color scale below the Color Shift surface, it is possible to estimate that the elevation of the minor peaks is probably somewhere between twenty-five and fifty percent of the elevation of the main central peak. It is clearly impossible to glean that kind of information from the Grayscale Plot of the same surface.

The highest elevation is obvious in the Color Shift Plot

By using white to indicate the highest elevation *(in addition to using shades of red to indicate high elevations)*, it is easy to determine the exact location of the highest elevation in the Color Shift Plot. The highest elevation is indicated by the small white square, which is on the cross hairs at the center of the plot.

While the highest elevation is pretty well indicated in the Grayscale Plot also, if the central peak were not symmetric in all four quadrants, there might be some uncertainty as to the exact location of the highest elevation.

Quantitative estimates are possible

In addition to providing a good overview of the shape of the 3D surface, the Color Shift Plot also makes it possible to estimate the elevations of points on the surface in a quantitative way. For example, in addition to black and white, the yellow and aqua colors are fairly easy to identify on the surface plot and are also fairly easy to identify on the calibration scale. In addition, the yellow and aqua bands on the calibration scale are fairly narrow. By measuring the locations of these colors on the calibration scale, the elevations of the yellow and aqua areas on the surface plot can be estimated with reasonable accuracy.

The ranges of the surface elevations colored red, green, and blue can also be estimated but with less certainty.

> *(Because the green portion of the calibration scale is about twice as wide as the red and blue portions, the level of uncertainty when using the green calibration data to estimate the elevation of a point on the surface is about twice the level of uncertainty when using the red or blue calibration data to estimate the elevation of a point on the surface.)*

Color Contour Plot

The image at the right in Figure 1 improves on the ability to provide good quantitative estimates of surface elevations.

If you need to read elevations off the surface plot to a high level of accuracy, the best approach is probably to use a Labeled Numeric Contour Plot. However, such plots are relatively difficult to create. Also, because of the need for the labels to be large enough to read, the space required to display such a plot can sometimes be excessive.

A reasonable compromise between a Labeled Numeric Contour Plot and a Color Shift Plot is the Color Contour Plot shown at the right end of Figure 1. This plotting format provides more accuracy in estimating surface elevations than the Color Shift Plot, but doesn't require any more space to display.

Similar to a contour map

The Color Contour Plot at the right in Figure 1 is similar to a contour map without labels on the contours. Each color traces out a constant elevation on the surface. The elevation indicated by a given color on the 3D surface can be determined by the position of that color in the calibration scale at the bottom of the image.

> *(This program quantizes the range from the lowest to the highest elevation into 23 levels. Therefore, the accuracy of an elevation estimate is good to only about one 23rd of that total range. However, it would be an easy matter to increase the number of quantization levels used in this program, thereby improving the accuracy of elevation estimates.)*

For example, the blue contour that surrounds the central peak traces out the shape of an elevation that is about three levels up from the lowest elevation *(as seen on the calibration scale).* The red contour that surrounds the central peak traces out the shape of an elevation that is about five levels down from the highest elevation. The aqua at the center of each of four minor peaks establishes their peak elevation to be about thirty-five percent of the elevation of the central peak *(based on the position of aqua in the calibration scale).*

More minor peaks

This plotting format also exposes four more minor peaks at the corners of the plot. The light gray color indicates that the level of these peaks is about two levels up from the lowest elevation. These four peaks are barely visible in the Color Shift Plot in the center, and their elevation is clearly not quantifiable in that plotting format. They are not visible at all in the Grayscale Plot on the left end of Figure 1.

The design of this program

There are numerous options available when designing a Color Contour Plotting program. As mentioned above, the Color Contour plotting format in this program subdivides the surface into 23 elevation levels including the lowest and the highest levels. Then it represents each elevation level with a different color or shade of gray. This causes a lot of quantitative information to become available that isn't available with either of the other two formats.

> *(There is nothing unique about 23 elevation levels and 23 colors. It would be very easy to use many more levels and many more colors. The biggest difficulty when designing the Color Contour format is identifying a large number of colors that are clearly identifiable both on the calibration scale and on the surface plot.)*

Good quantitative elevation information is available

To determine the elevation associated with a particular color, all you need to do is to locate that color on the calibration scale and determine its position relative to the colors at the ends. That will tell you the elevation associated with that color relative to the highest elevation and the lowest elevation. For example, the color at the exact center of the calibration scale represents an elevation that is half way between the lowest elevation and the highest elevation.

(With this program, there are no absolute elevations. Rather, the calibration scale indicates each elevation level as a percentage of the difference between the lowest and the highest elevations.)

The elevations of the minor peaks

Once again, this image shows that the elevations of the four minor peaks on the edges match the color aqua on the calibration scale. Judging from the position of the color aqua on the calibration scale, the elevation of each of the four minor peaks is about thirty-five percent of the elevation of the major peak in the center.

This information is clearly not available from the Grayscale Plot. It is also not available with this degree of accuracy from the Color Shift Plot.

(All that we can tell from the Color Shift Plot is that the minor peaks are some shade of green, which represents a rather large range of possible elevations.)

The lowest elevations

The Color Shift Plot does a better job of identifying the locations of the lowest elevations than does the Color Contour Plot. This is because the color black was dedicated to that purpose in the Color Shift Plot, but was used to represent a range of elevations in the Color Contour Plot.

(The color black could also be dedicated to identifying the lowest elevation in the design of the Color Contour Plot, in which case, both schemes would be equal in this regard.)

As we learned earlier, the lowest elevation occurs at four different points on the surface, and those points are near the corners.

The elevation of the valleys

Both plots show that while the valleys between the central and minor peaks are very deep, they aren't quite as deep as the lowest elevation. They are blue in the Color Shift Plot and gray in the Color Contour plot. Because the gray color represents a somewhat smaller elevation range in the Color Contour Plot than the blue represents in the Color Shift Plot, the elevation of the valley is defined more accurately in the Color Contour Plot.

A logarithmic conversion

Sometimes when plotting data, it is useful to plot the logarithm of the data values instead of the raw values.

(For many years, engineers have plotted data on graph paper referred to either as semi-log paper or log-log paper. Each type of graph paper has advantages and disadvantages relative to the other type and also has advantages and

disadvantages relative to linear graph paper. This program provides a capability that is analogous to the use of semi-log paper but in a 3D sense.)

Flattens the plot

The use of semi-log paper has the effect of flattening the plot in the 2D case, or flattening the surface in the 3D case. The semi-log approach tends to pull the structure of the low-level values up so that they can be better observed. The logarithm of the low elevations is closer to the maximum elevation than is the raw value of the low elevations.

More sample output

The three images in the top row of Figure 2 are reproductions of the three images in Figure 1. They were included in Figure 2 for comparison with the bottom three images in Figure 2.

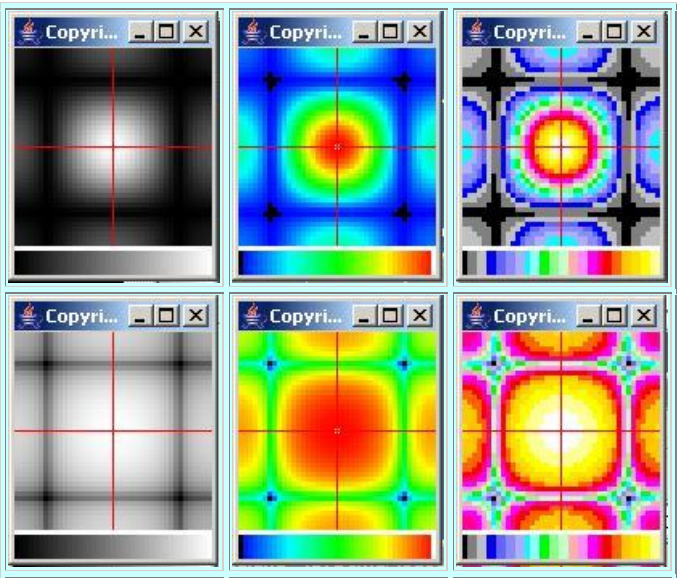

Figure 2

The bottom three images in Figure 2 were produced in exactly the same way as the top three images except that prior to creating the image the elevation values for the surface were converted to the log base 10 of the raw elevation values.

Six different formats

Thus, Figure 2 shows the same 3D surface plotted using six different plotting formats. Going from left to right and top to bottom, the six images illustrate:

- Grayscale *(linear)*
- Color Shift *(linear)*
- Color Contour *(linear)*
- Grayscale with logarithmic data conversion

- Color Shift with logarithmic data conversion
- Color Contour with logarithmic data conversion

Isolates the location of the minima

The significance of the logarithmic conversion can be seen by comparing the two images on the right side of Figure 2. When the raw elevation values were quantized into 23 levels, quite a few of the elevation values were quantized into the minimum value as indicated by the black areas in the top image.

However, after converting the elevation values to log values, only four points quantized to the minimum value as indicated by the four small black squares in the bottom right image. Thus, the top image on the right shows the general area of the lowest elevations on the surface whereas the bottom image on the right clearly identifies the exact location of each of the four lowest elevations.

A similar discussion holds regarding the two middle plots in Figure 2. The log version on the bottom identifies the location of the minima more closely than does the linear plot at the top.

The log of the surface is flatter

You can also see from the bottom right image of Figure 2 that the central peak of the log of the surface is much broader than the central peak for the raw surface at the top right. This is indicated by the width of the white and yellow areas in the log version as compared to the white and yellow areas in the raw version.

In addition, the elevations of the log data for the minor peaks at the four sides are almost as high as the elevation of the central peak, as indicated by the orange or yellow color at the top of the minor peaks.

The elevations of the tops of the four minor peaks at the corners are perhaps seventy-five percent of the elevation of the central peak as indicated by the pinkish color of those minor peaks in the log version.

> *(Recall that the elevation of the minor peaks at the corners is only about two levels up from the lowest elevation in the raw surface data.)*

All of this flattening was caused by converting the raw surface elevations to the log of the surface elevations before producing the surface plot.

Could quantize into more Color Contour levels

The quantization of the surface into 23 Color Contour levels was completely arbitrary. It would be easy to modify the program to quantize the surface into many more levels.

(The more difficult task would be to identify other colors that can be distinguished from one another and clearly identified when comparing surface plot colors to calibration scale colors. As you can see in Figure 2, some of the colors tend to run together even with only 23 colors. For example, I am unable to definitively match the "pinkish" color at the top of the minor peak in the bottom right image in Figure 2 with a specific color on the calibration scale.)

Multiple plotting formats are useful

A plotting format that works best for one surface doesn't necessarily work best for all surfaces. Therefore, it is useful for the program to be able to plot the same surface using different plotting formats.

Extremely easy to use

One of the main objectives in the development of this class was to make it very easy to use. No fancy programming is required to use the class and produce the plots. All that is required is to instantiate an object of the class named **ImgMod29**, passing an array of data to be plotted along with a few other parameters to the constructor. Basically the parameters *(in addition to the data array)* specify:

- Which of the three main formats to use, Grayscale, Color Shift, or Color Contour.
- Whether or not to convert to logarithmic values before plotting.
- Whether or not to draw the red axes shown in Figure 1 and Figure 2.
- How many pixels in the final output should be used to represent a single point on the surface. *(For example, the plots in Figure 1 and Figure 2 use a square of nine pixels to represent each point on the 3D surface. Note the small white square in the center of the center plot in Figure 1.)*

When an object of the class **ImgMod29** is instantiated, everything else happens automatically and the plot is displayed on the computer screen as illustrated by any one of the images in Figure 1 or Figure 2.

Multiple plots

If multiple plots in different formats are needed for a given set of data, all that is required is to instantiate multiple objects of the class named **ImgMod29** passing the same data array with different parameters to the constructor. Be aware, however, that all of the plots will be produced in a stack in the upper left corner of the screen. You must physically move the plots on the top in order to be able to view the plots lower down in the stack.

Preview

The purpose of the program that I will present and explain in this lesson is to display a 3D surface using color *(or shades of gray)* to represent the elevation of each point on a 3D surface.

The constructor for this class receives a 3D surface defined as a rectangular 2D array of values of type **double**. Each **double** value represents a sample point on the 3D surface. The surface values may be positive, negative, or both.

When an object of the class is constructed, it plots the 3D surface using one of six possible formats representing the elevation of each point on the surface with a color or a shade of gray. The constructor requires four parameters:

- double[][] **dataIn**
- int **blockSize**
- boolean **axis**
- int **display**

The purpose of each parameter is as follows:

dataIn

The parameter named **dataIn** is a reference to the 2D array of type **double** containing the data that describes the 3D surface.

blockSize

The value of the parameter named **blockSize** defines the size of a colored square in the final display that represents an input surface elevation value. For example, if **blockSize** is 1, each input surface value is represented by a single pixel in the display. If **blockSize** is 5, each input surface value is represented by a colored square having 5 pixels on each side.

The test code in the **main** method displays a surface having 59 values along the horizontal axis and 59 values along the vertical axis. Each elevation value on the surface is represented in the final display by a colored square that is 2 pixels on each side.

axis

The parameter named **axis** specifies whether optional red axes will be drawn on the display with the origin at the center as shown in Figure 1.

display

The parameter named **display** specifies one of six possible display formats. The value of **display** must be between 0 and 5 inclusive.

Values of 0, 1, and 2 specify the following formats:

display = 0

This value for the **display** parameter specifies a Grayscale Plot with a smooth gradient from black at the minimum to white at the maximum.

display = 1

This value for the **display** parameter specifies a Color Shift Plot with a smooth gradient from blue at the low end through aqua, green, and yellow to red at the high end. The minimum elevation is colored black. The maximum elevation is colored white.

display = 2

This value for the **display** parameter specifies a Color Contour Plot. The surface is subdivided into 23 levels and each of the 23 levels is represented by one of the following colors in order from lowest to highest elevation:

- Color.BLACK
- Color.GRAY
- Color.LIGHT_GRAY
- Color.BLUE
- new Color(100,100,255)
- new Color(140,140,255)
- new Color(175,175,255)
- Color.CYAN
- new Color(140,255,255)
- Color.GREEN
- new Color(140,255,140)
- new Color(200,255,200)
- Color.PINK
- new Color(255,140,255)
- Color.MAGENTA
- new Color(255,0,140)
- Color.RED
- new Color(255,100,0)
- Color.ORANGE
- new Color(255,225,0)
- Color.YELLOW
- new Color(255,255,150)
- Color.WHITE

Note that some of the colors in the above list refer to named color constants in the **Color** class. Others refer to new **Color** objects constructed by mixing the specified levels of red, green, and blue.

display = 3, 4, and 5

These values for the **display** parameter specify that the surface is to be plotted in the same format as for **display** values 1, 2, and 3, except that the surface elevation values are rectified *(made positive)* and converted to log base 10 before being represented by a color and plotted.

A calibration scale

When the surface is plotted, a horizontal calibration scale is plotted immediately below the surface plot showing the colors used in the surface plot. The colors begin with the color for the lowest elevation at the left and progress to the color for the highest elevation at the right.

Normalization

Regardless of whether the surface elevation values are first converted to log values or not, the surface values are normalized to cause them to extend from 0 to 255 before converting the elevation values to color and plotting them. The lowest elevation ends up with a value of 0. The highest elevation ends up with a value of 255.

For display value of 0 or 3

This is a Grayscale Plot or a log Grayscale Plot as shown at the left side of Figure 2. The highest normalized elevation with a value of 255 is painted white. The lowest normalized elevation with a value of 0 is painted black. The surface is represented using shades of gray.

The shade changes from black to white in a uniform gradient as the normalized surface elevation values progress from 0 to 255.

For display value of 1 or 4

This is a Color Shift Plot or log Color Shift Plot as shown in the center of Figure 2. The lowest normalized elevation is painted black and the highest normalized elevation is painted white. *(Black and white overwrite blue and red for these two elevation values.)*

The color changes from blue through aqua, green, and yellow to red in a smooth gradient as the normalized surface values progress from 1 to 254. *(Values of 0 and 255 would be pure blue and pure red if they were not painted black and white.)*

For a display value of 2 or 5

This is a Color Contour Plot or log Color Contour Plot as shown at the right side of Figure 2. The highest normalized elevation with a value of 255 is painted white. The lowest normalized elevation with a value of 0 is painted black.

The surface is represented using a combination of unique shades of gray and unique colors as the normalized surface elevation values progress from 0 to 255. This is not a gradient display. Rather, the colors in this display change abruptly from one color to the next.

This display format is similar to a contour map where each distinct color traces out a constant elevation level on the normalized surface being plotted.

The main method

Although the class is intended to be used by other programs to display surfaces produced by those programs, the class has a **main** method making it possible to run it in a stand-alone mode for testing.

When the class is run as a stand-alone program, it produces and displays six individual surfaces with the lowest point in the upper left corner and the highest point in the lower right corner. The six images produced by executing the **main** method are shown in Figure 3.

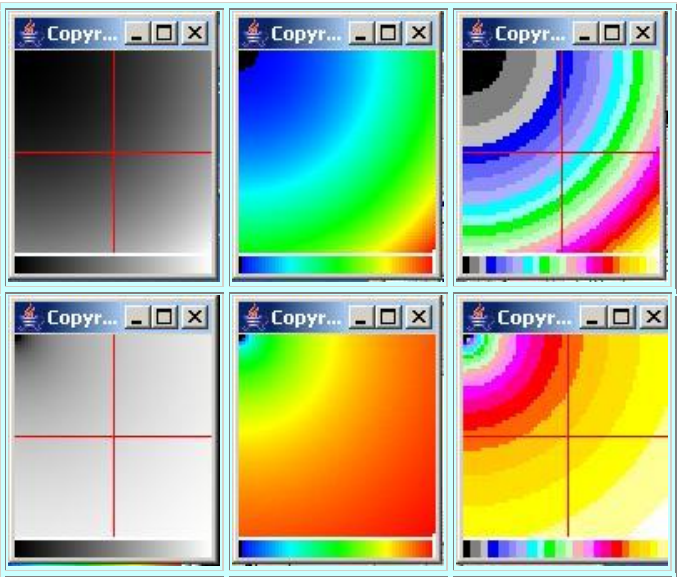

Figure 3

A 3D parabola

The **main** method creates and displays a surface consisting of a 3D parabola. You can think of the surface as representing a one-quarter section of a bowl, or perhaps a satellite dish with the center of the dish in the upper left corner of the image. The top three images in Figure 3 are the images produced from the raw surface. The bottom three images are the images produced by the log of the surface, *(which is no longer a 3D parabola).*

The calibration scale

The calibration scale is displayed immediately below the image of each surface.

The images are stacked

When the program is executed, the six surfaces are stacked in the upper left corner of the screen. *(You must physically move the images on the top to see the images on the bottom.)* The stacking order of the surfaces from bottom to top is based on the values of the **display** parameter in the order 0, 1, 2, 3, 4, 5.

With and without axes

Some of the surfaces show axes and some do not. This is controlled by the value of the constructor parameter named **axis**. A **true** value for **axis** causes the axes to be drawn.

A window listener

The constructor defines an anonymous inner class **WindowListener** on the close button on the **Frame** *(the X in the upper right hand corner of the **Frame**)*. Clicking the close button will terminate the program that uses an object of this class.

Testing

The program was tested using J2SE 5.0 and WinXP. Because the program uses Java features that were introduced in J2SE 5.0, it probably will not compile successfully with earlier versions of Java.

Sample Program

The program named ImgMod29

The program named **ImgMod29** is rather long, so as usual I will break it down and discuss it in fragments. You can view a complete listing of the program in Listing 29 near the end of the lesson.

The main method

The program consists of a top-level class named **ImgMod29**, plus several inner classes. To put things in context, I will begin my discussion with the **main** method, which is defined in the class named **ImgMod29**.

The **main** method begins in Listing 1.

```
  public static void main(String[]
args){
    int numberRows = 59;
    int numberCols = 59;
    double[][] data =
              new
double[numberRows][numberCols];
    int blockSize = 2;
```

```
Listing 1
```

The array for the surface elevation data

Listing 1 declares a 2D array of type **double** to contain the 3D surface elevation data values. This is a square array consisting of 59 elevation values on each side. *(However, there is no requirement for the surface to be square.)*

The blockSize parameter

Listing 1 also defines a value of 2 for the **blockSize** parameter. This variable will be passed to the constructor for the **ImgMod29** class, causing each elevation value to be plotted as a small square of four pixels, two pixels on each side of the square.

> *(The overall size of the display can be controlled by controlling the size of the array containing the surface elevation values and also controlling the value of **blockSize**.)*

A 3D parabolic surface

The array of surface elevation data is populated by the code in Listing 2.

```
    for(int row = 0;row <
numberRows;row++){
      for(int col = 0;col <
numberCols;col++){
        int xSquare = col * col;
        int ySquare = row * row;
        data[row][col] = xSquare +
ySquare;
      }//end col loop
    }//end row loop

Listing 2
```

I will allow you to evaluate this code on your own. It creates a 3D surface with the lowest elevation at the upper left corner and the highest elevation at the lower right corner. The surface is a one-quarter section of a 3D parabola, as shown in Figure 3.

The heart of the matter

Listing 3 shows the code that causes the 3D surface elevation data to be displayed as six independent images *(each statement in Listing 3 produces one output image).* This is the same code that would be used by other programs to incorporate this 3D surface plotting capability.

```

```

```
    new
ImgMod29(data,blockSize,true,0);
    new
ImgMod29(data,blockSize,false,1);
    new
ImgMod29(data,blockSize,true,2);
    new
ImgMod29(data,blockSize,true,3);
    new
ImgMod29(data,blockSize,false,4);
    new
ImgMod29(data,blockSize,true,5);
  }//end main

Listing 3
```

Just instantiate an object

All that is necessary for another program to incorporate this class to display a 3D surface is to instantiate an object of the class named **ImgMod29** passing four parameters to the constructor.

The parameters

The first parameter is a reference to the 2D array of type **double** containing the surface elevation values.

The second parameter is the **blockSize**. This **int** value specifies the size of one side of a square of pixels, *(all of the same color)* that will be used to represent each surface elevation value in the final display. As mentioned earlier, this value was set to 2 in the sample displays produced by the **main** method.

The third parameter is **true** if you want the red axes to be drawn and is **false** otherwise *(as shown in Figure 3).*

The fourth parameter is an integer value between 0 and 5 inclusive, which specifies the plotting format as follows:

- 0 - Grayscale *(linear)*
- 1 - Color Shift *(linear)*
- 2 - Color Contour *(linear)*
- 3 - Grayscale with logarithmic data conversion before plotting
- 4 - Color Shift with logarithmic data conversion before plotting
- 5 - Color Contour with logarithmic data conversion before plotting

Instantiate six separate objects

Listing 3 instantiates six such objects to display the same 3D surface, one for each plotting format as shown in Figure 3. Some of the objects display the axes and others do not. All use a **blockSize** value of 2.

Each display object appears in the upper-left corner of the screen. Thus, when two or more such objects are instantiated, they appear as a stack. It is necessary to physically move those on top of the stack to see those on the bottom.

Listing 3 signals the end of the **main** method.

The ImgMod29 class

Listing 4 shows the beginning of the class named **ImgMod29** and the beginning of the constructor for the class.

```
class ImgMod29 extends Frame{
  int dataWidth;
  int dataHeight;
  int blockSize;
  boolean axis;
  double[][] data;


  ImgMod29(double[][] dataIn,int
blockSize,
           boolean axis,int display){
    //Get and save several important
values
    this.blockSize = blockSize;
    this.axis = axis;
    dataHeight = dataIn.length;
    dataWidth = dataIn[0].length;
    boolean logPlot = false;
    int displayType = display;
```

Listing 4

The meaning and purpose of each of the constructor parameters was explained earlier, so I won't repeat that explanation here. The code in Listing 4 is straightforward and should not require further explanation.

Note that the constructor saves the value of the incoming parameter named **display** in the local variable named **displayType**. This establishes the default display format.

Re-establish display format for log conversion

The code in Listing 5 uses the incoming value of the **display** parameter to re-establish the display format if the value of **display** is 3, 4, or 5.

```
    if(display == 3){
      displayType = 0;
      logPlot = true;
    }else if(display == 4){
      displayType = 1;
      logPlot = true;
    }else if(display == 5){
      displayType = 2;
      logPlot = true;
    }else if((display > 5) || (display
< 0)){
      System.out.println(
         "DisplayType input error,
terminating");
      System.exit(0);
    }//end if
```

Listing 5

The default display format is one of the three basic types with no log conversion of the surface elevation data. If the incoming parameter value is 3, 4, or 5, the code in Listing 5 re-establishes the display format as one of the three basic types with log conversion of the surface elevation data prior to plotting.

> *(Note that the code in Listing 5 sets the value of the variable named* ***logPlot*** *to true. The value stored in this variable will be used later to determine if log conversion of the elevation data is required.)*

The three basic types

The three basic types are:

- displayType = 0, Grayscale Plot
- displayType = 1, Color Shift Plot
- displayType = 2, Color Contour Plot

These three basic types without log data conversion are shown from left to right in Figure 1. The three basic types are shown with log conversion from left to right in the bottom rows of Figure 2 and Figure 3.

Copy the input elevation data

Listing 6 makes a working copy of the input data to avoid damaging the original data. This is done to protect the data belonging to the program that instantiates an object of the class **ImgMod29**.

```
    data = new
```

```
double[dataHeight][dataWidth];
    for(int row = 0;row <
dataHeight;row++){
      for(int col = 0;col <
dataWidth;col++){
        data[row][col] =
dataIn[row][col];
      }//end loop on col
    }//end loop on row

Listing 6
```

Convert to log data if required

The code in Listing 7 uses the **log10** method of the **Math** class to perform a log conversion of the surface elevation data if the value of **logPlot** is true.

```
    if(logPlot){//Convert to log base
10.
      for(int row = 0;row <
dataHeight;row++){
        for(int col = 0;col <
dataWidth;col++){
          //Change the sign on
negative values
          // before converting to log
values.
          if(data[row][col] < 0){
            data[row][col] = -
data[row][col];
          }//end if
          if(data[row][col] > 0){
            //Convert value to log
base 10. Log
            // of 0 is undefined. Just
leave it
            // at 0.
            data[row][col] =

Math.log10(data[row][col]);
          }//end if
        }//end col loop
      }//end row loop
    }//end if on logPlot

Listing 7
```

New to J2SE 5.0

According to Sun's documentation, the **log10** method became part of Java with the release of J2SE 5.0. Therefore, this code is not compatible with earlier versions of Java.

Here are a couple of restrictions taken from Sun's documentation that apply to the use of the method named **log10**:

- If the argument is NaN or less than zero, then the result is NaN.
- If the argument is positive zero or negative zero, then the result is negative infinity.

Because of the first restriction, the code in Listing 7 converts all negative values into positive values before performing the conversion. Because of the second restriction, no attempt is made to compute the log of elevation values of zero.

Otherwise, the code in Listing 7 is straightforward and should not require further explanation.

Normalize the surface elevation data

After converting the elevation data to log form, *(or not converting as the case may be)*, the code in Listing 8 invokes the method named **scaleTheSurfaceData** to normalize the elevation data by squeezing it into the integer range between 0 and 255 inclusive. When this method returns, the lowest elevation has a value of 0 and the highest elevation has a value of 255.

```
    scaleTheSurfaceData();
```

Listing 8

The elevation data is normalized to this range to make it easier later to form relationships between the elevation values and allowable color values.

(Recall that allowable color values range from 0 to 255.)

The scaleTheSurfaceData method

The method named scaleTheSurfaceData is shown in its entirety in Listing 9.

```
  double min;
  double max;
  //This method is used to scale the
surface data
  // to force it to fit in the range
from 0 to
  // 255.
  void scaleTheSurfaceData(){
    //Find the minimum surface value.
    min = Double.MAX_VALUE;
    for(int row = 0;row <
dataHeight;row++){
      for(int col = 0;col <
dataWidth;col++){
        if(data[row][col] < min)
```

```
          min = data[row][col];
      }//end col loop
    }//end row loop

    //Shift all values up or down to
force new
    // minimum value to be 0.
    for(int row = 0;row <
dataHeight;row++){
      for(int col = 0;col <
dataWidth;col++){
        data[row][col] =
data[row][col] - min;
      }//end col loop
    }//end row loop

    //Now get the maximum value of the
shifted
    // surface values
    max = -Double.MAX_VALUE;
    for(int row = 0;row <
dataHeight;row++){
      for(int col = 0;col <
dataWidth;col++){
        if(data[row][col] > max)
          max = data[row][col];
      }//end col loop
    }//end row loop

    //Now scale all values to cause
the new
    // maximum value to be 255.
    for(int row = 0;row <
dataHeight;row++){
      for(int col = 0;col <
dataWidth;col++){
        data[row][col] =
                      data[row][col] *
255/max;
      }//end col loop
    }//end row loop
  }//end scaleTheSurfaceData

Listing 9
```

While this method is rather long, it is completely straightforward and shouldn't require further explanation.

Create an appropriate pair of Canvas objects

Return now to the discussion of the constructor for the **ImgMod29** class.

The code in Listing 10 uses the value of **displayType** to make a decision and to instantiate a pair of objects from two of six different inner classes, each of which extends the class named

Canvas. One of the objects in the pair is used to display the 3D surface according to a specified format. The other object in the pair is used to display the calibration strip below the surface display.

```
    Canvas surface = null;
    Canvas scale = null;

    //Establish the format based on
the value of
    // the parameter named display.
    if(displayType == 0){
      //Create a type 0 Canvas object
to draw the
      // surface on.  This is a
Grayscale Plot
      // display.
      surface = new
CanvasType0surface();
      //Create a Canvas object to draw
the scale
      // on for the GrayScale plot.
      scale = new CanvasType0scale();
    }else if(displayType == 1){
      //Color Shift Plot
      surface = new
CanvasType1surface();
      scale = new CanvasType1scale();
    }else if(displayType == 2){
      //Color Contour Plot.
      surface = new
CanvasType2surface();
      scale = new CanvasType2scale();
    }//end if-else on display type
```

Listing 10

The code in Listing 10 is straightforward and shouldn't require further explanation.

The interesting code

The interesting code is contained in the overridden **paint** method belonging to each of the six inner classes from which the pair of objects is instantiated. I will explain those overridden **paint** methods later.

Add the Canvas objects to the Frame

The default layout manager for a **Frame** object is **BorderLayout**. The code in Listing 11 adds one of the above-instantiated objects to the center location of the **Frame**, and adds the other object to the South location of the **Frame**. This produces the format with the surface plot above the calibration scale as shown by any of the images in Figure 1 through Figure 3.

```
    //Add the plotted surface to
center of the
    // Frame
    add(BorderLayout.CENTER,surface);
    //Add the scale to bottom of Frame
    add(BorderLayout.SOUTH,scale);
    //Cause the size of the Frame to
be just
    // right to contain the two Canvas
objects.
    pack();

    //Set Frame cosmetics and make it
visible.
    setTitle("Copyright 2005
R.G.Baldwin");
    setVisible(true);
```

Listing 11

Invoke the pack method and set the title

After adding the two objects to the **Frame**, Listing 11 invokes the **pack** method on the **Frame** to cause the size of the **Frame** to close in around the two objects.

Finally, Listing 11 sets the title on the **Frame** and makes it visible.

Register an anonymous WindowListener object

Listing 12 registers an anonymous **WindowListener** object on the **Frame** to cause the program to terminate whenever the user clicks the X-button in the upper right corner of the **Frame**.

```
    addWindowListener(new
WindowAdapter(){
        public void
windowClosing(WindowEvent e){
          System.exit(0);
        }//end windowClosing
      }//end class definition
    );//end addWindowListener

  }//end constructor
```

Listing 12

If you are unfamiliar with the use of anonymous inner classes, you can learn about such topics in my earlier lessons.

Listing 12 also signals the end of the constructor for the class named **ImgMod29**.

Six different inner classes

We have finally gotten to the fun part of the program. This is the part where we write the code that determines how the surface elevations and the calibration scale values are displayed. This part of the program consists of six different inner classes.

The fact that the classes are inner classes makes it possible for methods in the class to access instance variables and methods of the containing object of type **ImgMod29**. This makes the programming somewhat easier than would be the case if they were all top-level classes.

Subclasses of the Canvas class

Each of the six inner classes is a subclass of the class named **Canvas** and each of the classes overrides the **paint** method. The code in the overridden **paint** methods for the classes that display the 3D surfaces access the surface elevation data and convert that data into colors and display those colors in the correct locations on the screen. The names and behaviors of those three classes are:

- CanvasType0surface - Displays a Grayscale Plot
- CanvasType1surface - Displays a Color Shift Plot
- CanvasType2surface - Displays a Color Contour Plot

Associated directly with the three above classes are three other inner classes that are used to display the calibration scale immediately below the plot of the 3D surface. The names and behaviors of those three classes are:

- CanvasType0scale - Displays a Grayscale calibration scale
- CanvasType1scale - Displays a Color Shift calibration scale
- CanvasType2scale - Displays a Color Contour calibration scale

The getCenter method

Before getting into the details of these inner classes, however, I will present and briefly discuss a method named **getCenter**, which is invoked by the constructor for each of the surface plotting classes.

The **getCenter** method is used to find the horizontal and vertical center of the surface. These values are used to position the optional red axes that may be drawn on the surface. This method is shown in its entirety in Listing 13.

```
int horizCenter;
int vertCenter;

void getCenter(){
  if(dataWidth%2 == 0){//even
    horizCenter =
```

```
           dataWidth * blockSize/2 +
blockSize/2;
    }else{//odd
      horizCenter = dataWidth *
blockSize/2;
    }//end else

    if(dataHeight%2 == 0){//even
      vertCenter =
          dataHeight * blockSize/2 +
blockSize/2;
    }else{//odd
      vertCenter = dataHeight *
blockSize/2;
    }//end else
  }//end getCenter
```

Listing 13

Even or odd is very important

Note that the returned values depend on whether the dimensions of the surface are even or odd.

> *(For example, the center of a string of five blocks of pixels is the third block whereas the center of a string of six blocks of pixels is half way between the third and fourth blocks.)*

Now that you know about the difference between even and odd surface dimensions, the code in Listing 13 should be straightforward and should not require further discussion.

Grayscale Plot format: the class named CanvasType0surface

Listing 14 shows the beginning of the class named **CanvasType0surface** and also shows the entire constructor for that class.

An object of this class is used to paint a Grayscale Plot of a 3D surface ranging from black at the lowest elevation to white at the highest elevation. The various shades of gray vary in a smooth gradient between the two extremes. The leftmost image in Figure 1 is an example of the type of plot produced by an object of this class.

```
  class CanvasType0surface extends
Canvas{
    CanvasType0surface(){//constructor
      setSize(dataWidth * blockSize,
              dataHeight * blockSize);
      getCenter();
    }//end constructor
```

```
Listing 14
```

The constructor

The constructor for the class is straightforward. It accesses instance variables of the outer enclosing object to set the size of the canvas on the basis of the size of the surface and the size of the square of pixels that will be used to represent each elevation value on the surface.

The constructor also invokes the **getCenter** method to get the coordinates of the center of the surface in order to be able to draw the optional red axes in the correct position later.

Overridden paint method for CanvasType0surface class

The real work is done by the overridden **paint** method, which begins in Listing 15. Of the three classes used to plot the 3D surface, this is the simplest.

```
    public void paint(Graphics g){
      Color color = null;
      for(int row = 0;row <
dataHeight;row++){
        for(int col = 0;col <
dataWidth;col++){
          //Add in red, green, and
blue in
          // proportion to the value
of the
          // surface height.
          int red =
(int)data[row][col];
          int green = red;
          int blue = red;

Listing 15
```

Purpose of overridden paint method

The purpose of this overridden **paint** method is to convert the elevation value of each point on the 3D surface into an appropriate shade of gray, and to paint a square of pixels on the screen at that elevation value's location where every pixel in the square is the same shade of gray.

In order to produce a gray pixel in a 24-bit color system, you need to set the color values for red, green, and blue to the same value. If all three color values are zero, the pixel color is black. If all three color values are 255, the pixel color is white. If all three color values fall somewhere in between zero and 255, the pixel color will be some shade of gray.

Recall that the elevation values were earlier normalized to fall in the range from zero to 255. The code in Listing 15 uses a nested **for** loop to access every elevation value in the array

that describes the 3D surface. Then it sets the red, green, and blue color values to the normalized surface value for each point on the 3D surface.

Instantiate a Color object

Continuing inside the nested **for** loops, the code in Listing 16 instantiates a new object of type **Color** based on the current red, green, and blue color values.

```
          color = new
Color(red,green,blue);

Listing 16
```

Set the current drawing color

The code in Listing 17 invokes the **setColor** method of the **Graphics** class to set the current drawing color to the color described by the **Color** object referred to by the reference variable named **color**.

```
          //Set the color value.
          g.setColor(color);
          //Draw a square of the
specified size
          //in the specified color at
the
          // specified location.
          g.fillRect(col * blockSize,
                     row * blockSize,
                     blockSize,
                     blockSize);
        }//end col loop
      }//end row loop

Listing 17
```

Paint a colored square

Finally the code in Listing 17 invokes the **fillRect** method of the **Graphics** class to paint a square of pixels of the specified size at the specified location in the specified color.

This process is repeated for every elevation point on the 3D surface data, producing an output similar to the leftmost image in Figure 1.

Listing 17 signals the end of the nested **for** loops. When the code in Listing 17 finishes execution, the 3D surface has been plotted, but it does not yet contain the optional red axes.

Draw the optional red axes

Listing 18 tests to see if the value of the **axis** parameter is true. If so, it uses the information obtained earlier from the **getCenter** method, along with the **setColor** and **drawLine** methods of the **Graphics** class to draw the optional red axes shown in the images in Figure 1. These axes always intersect at the center of the image.

```
      if(axis){
        g.setColor(Color.RED);

g.drawLine(0,vertCenter,2*horizCenter,

vertCenter);

g.drawLine(horizCenter,0,horizCenter,

2*vertCenter);
      }//end if
    }//end paint
  }//end inner class
CanvasType0surface
```

Listing 18

Listing 18 also signals the end of the overridden **paint** method and the end of the inner class named CanvasType0Surface.

Plot the calibration scale for the Grayscale Plot format

An object of the class named **CanvasType0scale** is used to plot the calibration scale that is displayed immediately below the surface plot for the Grayscale Plot format. The beginning of this class and the constructor for this class are shown in Listing 19.

```
  class CanvasType0scale extends
Canvas{
    //Set the physical height of the
scale strip
    // in pixels.
    int scaleHeight = 6 * blockSize;

    CanvasType0scale(){//constructor
      //Set the size of the Canvas
based on the
      // width of the surface and the
size of the
      // square used to represent each
value on
      // the surface.
      setSize(dataWidth *
blockSize,scaleHeight);
    }//end constructor
```

Listing 19

How it works

Basically this class *(as well as the other two classes that create calibration scales)* operates by constructing an artificial surface, *(which is like a long thin board)*, positioned such that one end has an elevation of 0 and the other end has an elevation of 255. The length of this long thin surface is equal to the width of the surface plot for the Grayscale Plot format.

The same Grayscale color algorithm is applied to this artificial surface that is applied to the real surface. The result is a linear representation of the colors produced by the color algorithm from the lowest elevation at 0 to the highest elevation at 255. This result is displayed immediately below the real surface with the lowest elevation at the left end and the highest elevation at the right end. An example is shown in the leftmost image in Figure 1.

The code in Listing 19 establishes the size of the calibration scale surface.

The overridden paint method

Listing 20 shows the overridden **paint** method that is used to plot the calibration scale for the Grayscale Plot format.

```
    public void paint(Graphics g){
      //Vary from white to black going
from 255
      // to 0.
      Color color = null;
      //Don't draw in top row. Leave
it blank to
      // separate the scale strip from
the
      // drawing of the surface above
it.
      for(int row = 1;row <
scaleHeight;row++){
        for(int col = 0;col <
dataWidth;col++){

          //Compute the value of the
scale
          // surface.
          int scaleValue = 255 * col/

(dataWidth - 1);

          //See the class named
          // CanvasType0surface for
explanatory
          // comments regarding the
following
```

```
          // color algorithm.
          int red = scaleValue;
          int green = red;
          int blue = red;
          color = new
Color(red,green,blue);
          g.setColor(color);
          g.fillRect(col * blockSize,
                     row * blockSize,
                     blockSize,
                     blockSize);
        }//end col loop
      }//end row loop
    }//end paint

  }//end inner class CanvasType0scale
```

Listing 20

Because you already understand the color algorithm for the Grayscale Plot format, the code in Listing 20 should not require further explanation. This code establishes the elevation level for each point on the calibration surface and paints the box that represents that elevation in the appropriate color.

Color Shift Plot format: the class named CanvasType1surface

This class is used to instantiate an object that represents a normalized 3D surface with the colors ranging from blue at the low elevations through aqua, green, and yellow to red at the high elevations with a smooth gradient from 1 to 254.

> *(The lowest elevation with a value of 0 is colored black. The highest elevation with a value of 255 is colored white.)*

The center image in Figure 1 is an example of this plotting format.

The beginning of the class and the constructor

The beginning of the class and the constructor for the class are shown in Listing 21.

```
  class CanvasType1surface extends
Canvas{

    CanvasType1surface(){//constructor
      //Set the size of the Canvas
based on the
      // size of the surface and the
size of the
      // square used to represent each
value on
      // the surface.
```

```
      setSize(dataWidth * blockSize,
              dataHeight * blockSize);
      getCenter();
    }//end constructor

Listing 21
```

The beginning of all three classes that produce the three plotting formats is essentially the same. Therefore, the code in Listing 21 is essentially the same as the code in Listing 14 and should not require further explanation. The significant differences between the three classes lie in their overridden **paint** methods.

Overridden paint method for the CanvasType1surface class

The overridden **paint** method for this class, which begins in Listing 22, is probably the most complex of the three.

```
    public void paint(Graphics g){
      Color color = null;
      for(int row = 0;row <
dataHeight;row++){
        for(int col = 0;col <
dataWidth;col++){
          int red = 0;
          int green = 0;
          int blue = 0;

Listing 22
```

The **paint** method for this class begins by setting up a pair of nested **for** loops that will be used to process each elevation point on the surface, and by initializing the color values for red, green, and blue to 0 in the innermost loop.

How it works

If the elevation value is equal to 255, color values are set to cause that elevation to be painted white. If the elevation value is equal to 0, color values are set to cause that elevation to be painted black.

Listing 23 sets the color values to cause the extreme values of 0 and 255 to be painted black and white. Note that the code in Listing 23 is the beginning of a series of **if-else** constructs.

```
          if((int)data[row][col] ==
255){
            red = green = blue =
255;//white
          }else if((int)data[row][col]
```

```
== 0 ){
            red = green = blue =
0;//black

Listing 23
```

Elevations other than the extreme ends

If the elevation is not one of the extreme values of 0 or 255, control passes to code that subdivides the total elevation range from 1 to 254 into the following four ranges and then sets the color values for each range separately:

- 0 < elevation <= 63
- 63 < elevation <= 127
- 127 < elevation <= 191
- 191 < elevation <= 254

Processing 0 < elevation <= 63

Listing 24 shows the code that is used to process the lowest range of elevations between 1 and 63.

```
          }else
if(((int)data[row][col] > 0) &&

((int)data[row][col] <= 63)){
            int temp = 4 *
((int)data[row][col]

- 0);
            blue = 255;
            green = temp;

Listing 24
```

What we are shooting for here is to produce color values that will result in a smooth gradient of color from blue at the low end to aqua at the high end of the range.

> *(See the leftmost one-fourth of the calibration scale for the middle image in Figure 1.)*

Scale the elevation values

Listing 24 begins by multiplying the elevation value by a factor of 4 to put it into the range from 4 to 252. This makes the elevation values compatible with allowable color values that range from 0 to 255.

The color aqua

The color aqua is produced by mixing equal amounts of blue and green. Listing 24 holds the value of blue constant at 255 and increases the value of green in proportion to the elevation value. Thus, at the lower end of the range, blue has a value of 255 and green has a value of 4. *(This is almost pure blue.)* At the upper end of the range, blue still has a value of 255 and green has a value of 252. *(This is almost the pure secondary color aqua.)*

In all cases, the value of red is 0 within this range. These color values will be used later to instantiate a **Color** object, which will be used to control the plotting color for that portion of the display.

Processing the other three ranges

Now that you know the basic scheme, you shouldn't have any difficulty understanding the code for processing the other three ranges shown in Listing 25.

```
          }else
if(((int)data[row][col] > 63) &&

((int)data[row][col] <= 127)){
            int temp = 4 *
((int)data[row][col]

- 64);
            green = 255;
            blue = 255 - temp;

          }else
if(((int)data[row][col] > 127) &&

((int)data[row][col] <= 191)){
            int temp = 4 *
((int)data[row][col]

- 128);
            green = 255;
            red = temp;

          }else
if(((int)data[row][col] > 191) &&

((int)data[row][col] <= 254)){
            int temp = 4 *
((int)data[row][col]

- 192);
            red = 255;
            green = 255 - temp;

          }//end else
```

```
Listing 25
```

Gradient from aqua to green

The second range produces a smooth gradient from aqua to green. In this range, the green color value is held constant at 255 and the blue color value is caused to decrease in inverse proportion to the normalized color value.

Gradient from green to yellow

The third range produces a smooth gradient from green to yellow. *(Yellow is produced by mixing equal amounts of red and green.)* Within this range, green is held constant at a value of 255 and the value of red increases in direct proportion to the normalized elevation value.

Gradient from yellow to red

The fourth range produces a smooth gradient from yellow to red. Within this range, the value of red is held constant at 255 and the value of green decreases in inverse proportion to the normalized elevation value.

A homework assignment

A useful homework assignment would be for you to modify the program as follows:

> Subdivide the total range into eight sub ranges instead of four as I did. Choose four additional colors that you can produce by mixing various levels of red, green, and blue. Modify the code to cause the colors to vary with a smooth gradient through those eight colors in succession.

The rest of the overridden paint method

The rest of the overridden **paint** method for this class is essentially the same as the code that I explained in Listing 16 through Listing 18. Having set the current plotting color, the method goes on to paint a square of pixels in that color at the correct location. Then it draws the optional red axes if specified. Therefore, I won't repeat that explanation. You can view this code in Listing 29 near the end of the lesson.

The CanvasType1scale class

The inner class named **CanvasType1scale** is used to construct a color scale that matches the color algorithm used in the class named **CanvasType1surface**.

The overridden **paint** method for this class replicates the color algorithm in the overridden **paint** method for the **CanvasType1surface** class.

Except for the difference in the overridden **paint** method, the structure of this class is the same as the class named **CanvasType0scale**, which I discussed earlier beginning with Listing 19. Therefore, I won't repeat that discussion here. You can view the class in Listing 29 near the end of the lesson.

Color Contour Plot: the class named CanvasType2surface

The class named **CanvasType2surface** is an inner class used to instantiate an object that plots a surface where each elevation on the surface is represented by a color taken from a color palette containing a finite number of colors. As written, the color palette contains 23 different colors and shades of gray, but you can easily increase that number if you would like to do so.

The color palette

Before getting into the details of the class, I will explain the color palette. The color palette is produced by a method named **getColorPalette**, shown in its entirety in Listing 26. This is a utility method that is invoked by the inner classes named **CanvasType2surface** and **CanvasType2scale**.

```
  Color[] getColorPalette(){
    //Note that the following is an
initialized
    // 1D array of type Color.
    Color[] colorPalette = {
      Color.BLACK,//              0,
0,  0
      Color.GRAY,//
128,128,128
      Color.LIGHT_GRAY,//
192,192,192
      Color.BLUE,//               0,
0,255
      new
Color(100,100,255),//100,100,255
      new
Color(140,140,255),//140,140,255
      new
Color(175,175,255),//175,175,255
      Color.CYAN,//
0,255,255
      new
Color(140,255,255),//140,255,255
      Color.GREEN,//
0,255,  0
      new
Color(140,255,140),//140,255,140
      new
Color(200,255,200),//200,255,200
      Color.PINK,//
255,175,175
      new
```

```
Color(255,140,255),//255,140,255
      Color.MAGENTA,//         255,
0,255
      new Color(255,0,140),  //255,
0,140
      Color.RED,//             255,
0,  0
      new Color(255,100,0),//
255,100,  0
      Color.ORANGE,//
255,200,  0
      new Color(255,225,0),//
255,225,  0
      Color.YELLOW,//
255,255,  0
      new
Color(255,255,150),//255,255,150
      Color.WHITE};//
255,255,255

    return colorPalette;
  }//end getColorPalette

Listing 26
```

The getColorPalette method

The purpose of this method is to establish a color palette containing references to **Color** objects representing 23 distinct colors and shades of gray. The references are stored in a one-dimensional array object as element type **Color**. The values shown in comments in Listing 26 represent the values of red, green, and blue required to produce that specific color.

As you can see, some of the elements in the array refer to **Color** objects defined as named constants *(public final variables)* in the **Color** class. Other elements in the array refer to **Color** objects that are instantiated using red, green, and blue color values of my own choosing. The actual colors represented by these objects, going from top to bottom, match the colors shown in the calibration scale for the rightmost image in Figure 1.

Rearrange and add new colors

If you would like to do so, you can rearrange the colors in the array. This will result in different colors being adjacent to one another in the calibration scale. Also if you would like to do so, you can remove colors from the array or add new colors of your own choosing to the array. The overridden **paint** methods in the classes named **CanvasType2surface** and **CanvasType2scale** are designed to take such changes into account.

The CanvasType2surface class

You can view the entire class named **CanvasType2surface** in Listing 29 near the end of the lesson. Because of the similarity of this class to others that I have previously discussed, I will

limit my discussion to the portions of the overridden **paint** method that distinguishes this class from the others.

As it turns out, this is perhaps the simplest of the three overridden **paint** methods. The method begins in Listing 27 where the **getColorPalette** method is called to get a reference to the color palette discussed above.

Then a pair of nested **for** loops is set up to process every elevation value on the 3D surface.

```
    public void paint(Graphics g){
      Color[] colorPalette =
getColorPalette();

      for(int row = 0;row <
dataHeight;row++){
        for(int col = 0;col <
dataWidth;col++){

          int quantizedData =
(int)(Math.round(
                  data[row][col]*(

colorPalette.length-1)/255));
```

Listing 27

Quantize the elevation levels

The code in Listing 27 quantizes the elevation levels into a set of integer values ranging from 0 to one less than the number of elements in the color palette. As written, this redefines the normalized elevation values as extending from 0 to 22, instead of from 0 to 255.

> *(If you change the **length** of the color palette, the number of ranges will change accordingly.)*

Set the color value

The code in Listing 28 uses the quantized elevation value to index into the color palette and retrieve a reference to a **Color** object. This reference is passed to the **setColor** method setting the current plotting color to the color represented by that index value.

```
          g.setColor(colorPalette[

quantizedData]);
```

Listing 28

Having set the current plotting color, the method goes on to paint a square of pixels in that color at the correct location. Then it draws the optional axes if specified. The code to accomplish these operations is the same as code discussed previously, so I won't repeat that discussion here. You can view the code in Listing 29 near the end of the lesson.

The class named CanvasType2scale

This inner class is used to construct a color scale that matches the color algorithm used in the class named **CanvasType2surface**. Except for the difference in the overridden **paint** method, this class is essentially the same as the other two classes used to construct color scale objects. Therefore, I won't repeat that discussion.

You can view the class in its entirety in Listing 29 near the end of the lesson. You can view the graphic output produced by this class in the calibration scale for the image at the rightmost end of Figure 1.

Run the Program

I encourage you to copy, compile, and run the program that you will find in Listing 29 near the end of the lesson. Modify the program and experiment with it in order to learn as much as you can about the use of Java for displaying 3D data.

A better color scheme

See if you can come up with a better color scheme than the color schemes that I used in my version of the program. For example, you might add new colors to the color palette used for the Color Contour Plot. That will be very easy to do. All you need to do is add them to the array.

> *(The hard part will be to identify new colors that are visually separable from the colors that are already being used.)*

You might also add new colors to the color algorithm for the Color Shift Plot. This will be somewhat more difficult in that additional coding will be required to incorporate those new colors.

Create different test surfaces

You might also want to modify the code in the **main** method to cause it to create different test surfaces. You could even write new independent programs that create surfaces and use this class named **ImgMod29** to plot those surfaces. Remember, all that's necessary to use this class to plot your own 3D surface is to include a statement similar to the following in your code:

```
new ImgMod29(data,blockSize,true,0);
```

Create larger test surfaces with a smaller blockSize

It was necessary for me to keep the images in this lesson small in order to force them to fit into this narrow publication format. As you are experimenting, make your test surfaces larger and your **blockSize** smaller. This will result in smoother edges where different colors meet.

The parameters to the ImgMod29 constructor

The parameter named **data** in the above example is a reference to a 2D array of type **double** that describes the surface to be plotted.

The second parameter named **blockSize** specifies the size of one side of the square of pixels in the final plot that you want to use to represent each elevation point on your 3D surface. Set this to 0 if you are unsure as to what size square you need.

The third parameter specifies whether or not you want to have the optional red axes drawn. A value of **true** causes the axes to be drawn. A value of **false** causes the axes to be omitted.

The fourth parameter is an integer that specifies the plotting format as follows:

- 0 - Grayscale *(linear)*
- 1 - Color Shift *(linear)*
- 2 - Color Contour *(linear)*
- 3 - Grayscale with logarithmic data conversion
- 4 - Color Shift with logarithmic data conversion
- 5 - Color Contour with logarithmic data conversion

The class couldn't be simpler to use.

Above all, have fun and learn as much as you can in the process.

What's Next?

I will be using this 3D plotting program in a variety of future lessons involving such complex topics as the use of the 2D Fourier Transform to process images and the embedding of secret watermarks in images.

Summary

In this lesson, I explained and illustrated a program for using Java and color to plot 3D surfaces. The program is extremely easy to use and makes it easy to plot your surface using six different plotting formats in a wide range of sizes.

Complete Program Listings

A complete listing of the program is provided in Listing 29 below.

```
/*File ImgMod29.java.java
Copyright 2005, R.G.Baldwin

The purpose of this program is to display a 3D
surface using color to represent the height of
each point on the surface.

The constructor for this class receives a 3D
surface defined as a rectangular 2D array of
double values.  The surface values may be
positive or negative or both.  When an object of
the class is constructed, it draws the 3D surface
using one of six possible formats representing
the height of each point on the surface with a
color.

The constructor requires four parameters:

double[][] dataIn
int blockSize
boolean axis
int display

The purpose of each parameter is as follows:

dataIn - The parameter named dataIn is a
reference to the array containing the data that
describes the 3D surface.

blockSize - The value of the parameter named
blockSize defines the size of a colored square in
the final display that represents an input
surface value.  For example, if blockSize is 1,
each input surface value will be represented by a
single pixel in the display.  If blockSize is 5,
each input surface value will be represented by
a colored square having 5 pixels on each side.
For example, the test code in the main method
displays a surface having 59 values along the
horizontal axis and 59 values along the vertical
axis.  Each value on the surface is represented
in the final display by a colored square that is
2 pixels on each side.

axis - The parameter named axis specifies whether
red axes will be drawn on the display with the
origin at the center.

display - The parameter named display specifies
one of six possible display formats.  The value
of display must be between 0 and 5 inclusive.
Values of 0, 1, and 2 specify the following
formats:

  0 - Gray scale gradient from black at the
```

```
  minimum to white at the maximum.
  1 - Color gradient from blue at the low end
  through aqua, green, yellow to red at the
  high end. The minimum value is colored black.
  The maximum value is colored white..
  2.- The surface is subdivided into 23 levels
  and each of the 23 levels is represented by
  one of the following Color Contour Plot in
order
  from minimum to maximum.
    Color.BLACK
    Color.GRAY
    Color.LIGHT_GRAY
    Color.BLUE
    new Color(100,100,255)
    new Color(140,140,255)
    new Color(175,175,255)
    Color.CYAN
    new Color(140,255,255)
    Color.GREEN
    new Color(140,255,140)
    new Color(200,255,200)
    Color.PINK
    new Color(255,140,255)
    Color.MAGENTA
    new Color(255,0,140)
    Color.RED
    new Color(255,100,0)
    Color.ORANGE
    new Color(255,225,0)
    Color.YELLOW
    new Color(255,255,150)
    Color.WHITE

Values of 3, 4, and 5 for the parameter named
display draw the surface in the same formats as
above except that the surface values are first
rectified and then converted to log base 10
values before being converted to color and drawn.

When the surface is drawn, a horizontal scale
strip is drawn immediately below the surface
showing the colors used in the drawing
starting with the color for the minimum at the
left and progressing to the color for the maximum
at the right.

Regardless of whether the surface values are
converted to log values or not, the surface
values are normalized to cause them to extend
from 0 to 255 before converting to color and
drawing.

For a display value of 0 or 3, the highest point
with a value of 255 is painted white.  The lowest
point with a value of 0 is painted black,  The
```

```
surface is represented using shades of gray.  The
shade changes from black to white in a uniform
gradient as the height of the normalized surface
values progress from 0 to 255.

For a display value of 1 or 4, the lowest point
is painted black and the highest point is
painted white. The color changes from blue
through aqua, green, and yellow to red in a
smooth gradient as the normalized surface values
progress from 1 to 254.  (Values of 0 and 255
would be pure blue and pure red if they were not
overridden by black and white.)

For a display value of 2 or 5, the highest point
with a value of 255 is painted white.  The lowest
point with a value of 0 is painted black,  The
surface is represented using a combination of
unique shades of gray and unique colors as the
normalized surface values progress from 0 to 255.
This is not a gradient display.  Rather, this
display format is similar to a contour map where
each distinct color traces out a constant level
on the normalized surface being drawn.

Although the class is intended to be used by
other programs to display surfaces produced by
those programs, the class has a main method
making it possible to run it in a stand-alone
mode for testing.  When run as a stand-alone
program, the class produces and displays six
individual surfaces with the lowest point in the
upper left corner and the highest point in the
lower right corner.  The scale strip is displayed
immediately below each surface. The six surfaces
are stacked in the upper left corner of the
screen.  (You must physically move the ones on
the top to see the ones on the bottom.)  The
stacking order of the surfaces from bottom to top
is based on display types in the order 0, 1, 2,
3, 4, and 5.  The surfaces that are displayed
are 3D parabolas. Some of the surfaces show axes
and some do not.

The constructor defines an anonymous inner class
listener on the close button on the frame.
Clicking the close button will terminate the
program that uses an object of this class.

Tested using J2SE 5.0 and WinXP
**********************************************/
import java.awt.*;
import java.awt.event.*;

class ImgMod29 extends Frame{
  int dataWidth;
```

```
int dataHeight;
int blockSize;
boolean axis;
double[][] data;


ImgMod29(double[][] dataIn,int blockSize,
         boolean axis,int display){
  //Get and save several important values
  this.blockSize = blockSize;
  this.axis = axis;
  dataHeight = dataIn.length;
  dataWidth = dataIn[0].length;
  boolean logPlot = false;
  int displayType = display;

  //Plot types 0, 1, and 2 with no log
  // conversion for display parameter
  // value  = 0, 1, or 2. This is the default
  // and no special code is required.
  //Plot types 0, 1, and 2 with log conversion
  // for display parameter value = 3, 4, or 5.
  if(display == 3){
    displayType = 0;
    logPlot = true;
  }else if(display == 4){
    displayType = 1;
    logPlot = true;
  }else if(display == 5){
    displayType = 2;
    logPlot = true;
  }else if((display > 5) || (display < 0)){
    System.out.println(
       "DisplayType input error, terminating");
    System.exit(0);
  }//end if

  //Make a copy of the input data array to
  // avoid damaging the original data.
  data = new double[dataHeight][dataWidth];
  for(int row = 0;row < dataHeight;row++){
    for(int col = 0;col < dataWidth;col++){
      data[row][col] = dataIn[row][col];
    }//end loop on col
  }//end loop on row

  if(logPlot){//Convert to log base 10.
    for(int row = 0;row < dataHeight;row++){
      for(int col = 0;col < dataWidth;col++){
        //Change the sign on negative values
        // before converting to log values.
        if(data[row][col] < 0){
          data[row][col] = -data[row][col];
        }//end if
        if(data[row][col] > 0){
          //Convert value to log base 10. Log
```

```
          // of 0 is undefined. Just leave it
          // at 0.
          data[row][col] =
                    Math.log10(data[row][col]);
        }//end if
      }//end col loop
    }//end row loop
  }//end if on logPlot

  //Force the data into the range from 0 to 255
  // regardless of whether or not it has been
  // converted to log values.
  scaleTheSurfaceData();

  Canvas surface = null;
  Canvas scale = null;

  //Establish the format based on the value of
  // the parameter named display.
  if(displayType == 0){
    //Create a type 0 Canvas object to draw the
    // surface on.  This is a gray scale
    // display.
    surface = new CanvasType0surface();
    //Create a Canvas object to draw the scale
    // on.
    scale = new CanvasType0scale();
  }else if(displayType == 1){
    //Color Shift Plot
    surface = new CanvasType1surface();
    scale = new CanvasType1scale();
  }else if(displayType == 2){
    //Color Contour Plot.
    surface = new CanvasType2surface();
    scale = new CanvasType2scale();
  }//end if-else on display type

  //Add the plotted surface to center of the
  // Frame
  add(BorderLayout.CENTER,surface);
  //Add the scale to bottom of Frame
  add(BorderLayout.SOUTH,scale);
  //Cause the size of the Frame to be just
  // right to contain the two Canvas objects.
  pack();

  //Set Frame cosmetics and make it visible.
  setTitle("Copyright 2005 R.G.Baldwin");
  setVisible(true);

  //Use an anonymous class to register a window
  // listener on the Frame.  This class extends
  // WindowAdapter
  addWindowListener(new WindowAdapter(){
      public void windowClosing(WindowEvent e){
        System.exit(0);
```

```
      }//end windowClosing
    }//end class definition
  );//end addWindowListener

}//end constructor
//-----------------------------------------//

double min;
double max;
//This method is used to scale the surface data
// to force it to fit in the range from 0 to
// 255.
void scaleTheSurfaceData(){
  //Find the minimum surface value.
  min = Double.MAX_VALUE;
  for(int row = 0;row < dataHeight;row++){
    for(int col = 0;col < dataWidth;col++){
      if(data[row][col] < min)
        min = data[row][col];
    }//end col loop
  }//end row loop

  //Shift all values up or down to force new
  // minimum value to be 0.
  for(int row = 0;row < dataHeight;row++){
    for(int col = 0;col < dataWidth;col++){
      data[row][col] = data[row][col] - min;
    }//end col loop
  }//end row loop

  //Now get the maximum value of the shifted
  // surface values
  max = -Double.MAX_VALUE;
  for(int row = 0;row < dataHeight;row++){
    for(int col = 0;col < dataWidth;col++){
      if(data[row][col] > max)
        max = data[row][col];
    }//end col loop
  }//end row loop

  //Now scale all values to cause the new
  // maximum value to be 255.
  for(int row = 0;row < dataHeight;row++){
    for(int col = 0;col < dataWidth;col++){
      data[row][col] =
                    data[row][col] * 255/max;
    }//end col loop
  }//end row loop
}//end scaleTheSurfaceData
//-----------------------------------------//

//main method for self-testing the class
public static void main(String[] args){
  //Create the array of test data.
  int numberRows = 59;
  int numberCols = 59;
```

```
    double[][] data =
              new double[numberRows][numberCols];
    int blockSize = 2;

    //Create a surface with a minimum at the
    // upper left corner and a maximum at the
    // lower right corner.  This surface is
    // a 3D parabola.
    for(int row = 0;row < numberRows;row++){
      for(int col = 0;col < numberCols;col++){
        int xSquare = col * col;
        int ySquare = row * row;
        data[row][col] = xSquare + ySquare;
      }//end col loop
    }//end row loop

    //Instantiate objects to display the test
    // data surface in six different formats on
    // top of one another in the upper left
    // corner of the screen.  Represent each
    // surface value by a colored square that is
    // blockSize pixels on each side.  Draw a red
    // axis at the center of some of the
    // surfaces.
    new ImgMod29(data,blockSize,true,0);
    new ImgMod29(data,blockSize,false,1);
    new ImgMod29(data,blockSize,true,2);
    new ImgMod29(data,blockSize,true,3);
    new ImgMod29(data,blockSize,false,4);
    new ImgMod29(data,blockSize,true,5);
  }//end main
  //---------------------------------------//

  int horizCenter;
  int vertCenter;
  //This helper method is used to find the
  // horizontal and vertical center of the
  // surface.  These values are used to locate
  // the red axes that are drawn on the surface.
  // Note that the returned values depend on
  // whether the dimensions of the surface are
  // odd or even.
  void getCenter(){
    if(dataWidth%2 == 0){//even
      horizCenter =
           dataWidth * blockSize/2 + blockSize/2;
    }else{//odd
      horizCenter = dataWidth * blockSize/2;
    }//end else

    if(dataHeight%2 == 0){//even
      vertCenter =
          dataHeight * blockSize/2 + blockSize/2;
    }else{//odd
      vertCenter = dataHeight * blockSize/2;
    }//end else
```

```
  }//end getCenter
  //----------------------------------------//

  //Note that the following six classes are
  // inner classes. This makes it possible for
  // methods in the class to access instance
  // variables and methods of the containing
  // object.

  //This class is used to draw a gray scale
  // surface ranging from white at the high end
  // to black at the low end with a smooth
  // gradient in between.
  class CanvasType0surface extends Canvas{
    CanvasType0surface(){//constructor
      //Set the size of the Canvas based on the
      // size of the surface and the size of the
      // square used to represent each value on
      // the surface.
      setSize(dataWidth * blockSize,
              dataHeight * blockSize);
      getCenter();
    }//end constructor

    //Override the paint method to draw the
    // surface.
    public void paint(Graphics g){
      //Vary from white to black going from high
      // to low.
      Color color = null;
      for(int row = 0;row < dataHeight;row++){
        for(int col = 0;col < dataWidth;col++){
          //Add in red, green, and blue in
          // proportion to the value of the
          // surface height.
          int red = (int)data[row][col];
          int green = red;
          int blue = red;
          //Compute the color value for the
          // point on the surface.
          color = new Color(red,green,blue);
          //Set the color value.
          g.setColor(color);
          //Draw a square of the specified size
          //in the specified color at the
          // specified location.
          g.fillRect(col * blockSize,
                     row * blockSize,
                     blockSize,
                     blockSize);
        }//end col loop
      }//end row loop

      //If axis is true, draw red lines to form
      // an origin at the center
      if(axis){
```

```
      g.setColor(Color.RED);
      g.drawLine(0,vertCenter,2*horizCenter,
                                    vertCenter);
      g.drawLine(horizCenter,0,horizCenter,
                                   2*vertCenter);
    }//end if
  }//end paint
}//end inner class CanvasType0surface
//=========================================//

//Note that this is an inner class.
//This class is used to construct a color scale
// that matches the color scheme used in the
// class named CanvasType0surface.
class CanvasType0scale extends Canvas{
  //Set the physical height of the scale strip
  // in pixels.
  int scaleHeight = 6 * blockSize;

  CanvasType0scale(){//constructor
    //Set the size of the Canvas based on the
    // width of the surface and the size of the
    // square used to represent each value on
    // the surface.
    setSize(dataWidth * blockSize,scaleHeight);
  }//end constructor

  //Override the paint method to draw the
  // scale strip.
  public void paint(Graphics g){
    //Vary from white to black going from 255
    // to 0.
    Color color = null;
    //Don't draw in top row. Leave it blank to
    // separate the scale strip from the
    // drawing of the surface above it.
    for(int row = 1;row < scaleHeight;row++){
      for(int col = 0;col < dataWidth;col++){

        //Compute the value of the scale
        // surface.
        int scaleValue = 255 * col/
                             (dataWidth - 1);

        //See the class named
        // CanvasType0surface for explanatory
        // comments regarding the following
        // color algorithm.
        int red = scaleValue;
        int green = red;
        int blue = red;
        color = new Color(red,green,blue);
        g.setColor(color);
        g.fillRect(col * blockSize,
                   row * blockSize,
                   blockSize,
```

```
                    blockSize);
      }//end col loop
    }//end row loop
  }//end paint

}//end inner class CanvasType0scale
//=========================================//

//This class is used to draw a surface with the
// colors ranging from blue at the low end
// through aqua, green, and yellow to red at
// the high end with a smooth gradient from 1
// to 254.  The lowest point with a value of 0
// is colored black.  The highest point with a
// value of 255 is colored white.
class CanvasType1surface extends Canvas{

  CanvasType1surface(){//constructor
    //Set the size of the Canvas based on the
    // size of the surface and the size of the
    // square used to represent each value on
    // the surface.
    setSize(dataWidth * blockSize,
            dataHeight * blockSize);
    getCenter();
  }//end constructor

  //Override the paint method to draw the
  // surface.
  public void paint(Graphics g){
    //Vary color as described in the comments
    // above.
    Color color = null;
    for(int row = 0;row < dataHeight;row++){
      for(int col = 0;col < dataWidth;col++){
        int red = 0;
        int green = 0;
        int blue = 0;

        if((int)data[row][col] == 255){
          red = green = blue = 255;//white
        }else if((int)data[row][col] == 0 ){
          red = green = blue = 0;//black

        }else if(((int)data[row][col] > 0) &&
                 ((int)data[row][col] <= 63)){
          int temp = 4 * ((int)data[row][col]
                                          - 0);
          blue = 255;
          green = temp;

        }else if(((int)data[row][col] > 63) &&
                ((int)data[row][col] <= 127)){
          int temp = 4 * ((int)data[row][col]
                                          - 64);
          green = 255;
```

```
          blue = 255 - temp;

        }else if(((int)data[row][col] > 127) &&
                 ((int)data[row][col] <= 191)){
          int temp = 4 * ((int)data[row][col]
                                        - 128);
          green = 255;
          red = temp;

        }else if(((int)data[row][col] > 191) &&
                 ((int)data[row][col] <= 254)){
          int temp = 4 * ((int)data[row][col]
                                        - 192);
          red = 255;
          green = 255 - temp;

        }else{//impossible condition
          System.out.println(
                    "Should not reach here.");
          System.exit(0);
        }//end else

        //Compute the color value for the
        // point on the surface.
        color = new Color(red,green,blue);
        //Set the color value.
        g.setColor(color);
        //Draw a square of the specified size
        // in the specified color at the
        // specified location.
        g.fillRect(col * blockSize,
                   row * blockSize,
                   blockSize,
                   blockSize);
      }//end col loop
    }//end row loop

    //If axis is true, draw red lines to form
    // an origin at the center
    if(axis){
      g.setColor(Color.RED);
      g.drawLine(0,vertCenter,2*horizCenter,
                                    vertCenter);
      g.drawLine(horizCenter,0,horizCenter,
                                  2*vertCenter);
    }//end if
  }//end paint
}//end inner class CanvasType1surface
//=========================================//

//Note that this is an inner class.  This class
// is used to construct a color scale that
// matches the color scheme used in the class
// named CanvasType1surface.
class CanvasType1scale extends Canvas{
 int scaleHeight = 6 * blockSize;
```

```
  CanvasType1scale(){//constructor
    //Set the size of the Canvas based on the
    // width of the surface and the size of the
    // square used to represent each value on
    // the surface.
    setSize(dataWidth * blockSize,scaleHeight);
  }//end constructor

  //Override the paint method to draw the
  // scale.
  public void paint(Graphics g){
    //Vary from yellow to blue going from 255
    // to 0.
    Color color = null;
    for(int row = 1;row < scaleHeight;row++){
      for(int col = 0;col < dataWidth;col++){

        int scaleValue = 255 * col/(
                                 dataWidth - 1);
        // See the class named
        // CanvasType1surface for explanatory
        // comments regarding this color
        // algorithm.
        int red = 0;
        int green = 0;
        int blue = 0;
        if(scaleValue == 255){
          red = green = blue = 255;//white
        }else if(scaleValue == 0 ){
          red = green = blue = 0;//black

        }else if((scaleValue > 0) &&
                            (scaleValue <= 63)){
          scaleValue = 4 * (scaleValue - 0);
          blue = 255;
          green = scaleValue;

        }else if((scaleValue > 63) &&
                           (scaleValue <= 127)){
          scaleValue = 4 * (scaleValue - 64);
          green = 255;
          blue = 255 - scaleValue;

        }else if((scaleValue > 127) &&
                           (scaleValue <= 191)){
          scaleValue = 4 * (scaleValue - 128);
          green = 255;
          red = scaleValue;

        }else if((scaleValue > 191) &&
                           (scaleValue <= 254)){
          scaleValue = 4 * (scaleValue - 192);
          red = 255;
          green = 255 - scaleValue;
```

```
        }else{//impossible condition
          System.out.println(
                    "Should not reach here.");
          System.exit(0);
        }//end else

        color = new Color(red,green,blue);
        g.setColor(color);
        g.fillRect(col * blockSize,
                   row * blockSize,
                   blockSize,
                   blockSize);
      }//end col loop
    }//end row loop
  }//end paint

}//end inner class CanvasType1scale
//=========================================//

//This is a utility method used by the two
// inner classes that follow.  The purpose of
// this method is to establish a color palette
// containing 23 distinct Colors and shades of
// gray.  The values shown in comments
// represent the values of red, green, and blue
// for that specific color.
Color[] getColorPalette(){
  //Note that the following is an initialized
  // 1D array of type Color.
  Color[] colorPalette = {
    Color.BLACK,//              0,  0,  0
    Color.GRAY,//            128,128,128
    Color.LIGHT_GRAY,//      192,192,192
    Color.BLUE,//              0,  0,255
    new Color(100,100,255),//100,100,255
    new Color(140,140,255),//140,140,255
    new Color(175,175,255),//175,175,255
    Color.CYAN,//              0,255,255
    new Color(140,255,255),//140,255,255
    Color.GREEN,//             0,255,  0
    new Color(140,255,140),//140,255,140
    new Color(200,255,200),//200,255,200
    Color.PINK,//            255,175,175
    new Color(255,140,255),//255,140,255
    Color.MAGENTA,//         255,  0,255
    new Color(255,0,140),  //255,  0,140
    Color.RED,//             255,  0,  0
    new Color(255,100,0),//  255,100,  0
    Color.ORANGE,//          255,200,  0
    new Color(255,225,0),//  255,225,  0
    Color.YELLOW,//          255,255,  0
    new Color(255,255,150),//255,255,150
    Color.WHITE};//          255,255,255

  return colorPalette;
}//end getColorPalette
```

```
  //=======================================//

  //Note that this is an inner class.
  //This class is used to draw a surface
  // representing the heights of the points on
  // the surface using the colors and shades of
  // gray defined in the color palette..
  class CanvasType2surface extends Canvas{

    CanvasType2surface(){//constructor
      //Set the size of the Canvas based on the
      // size of the surface and the size of the
      // square used to represent each value on
      // the surface.
      setSize(dataWidth * blockSize,
              dataHeight * blockSize);
      getCenter();
    }//end constructor

    //Override the paint method to draw the
    // surface.
    public void paint(Graphics g){
      Color[] colorPalette = getColorPalette();

      for(int row = 0;row < dataHeight;row++){
        for(int col = 0;col < dataWidth;col++){
          //Quantize the surface into a set of
          // levels where the number of levels is
          // equal to the number of colors in the
          // color palette.
          int quantizedData = (int)(Math.round(
                  data[row][col]*(
                    colorPalette.length-1)/255));
          //Set the color for this point to the
          // corresponding color from the
          // palette by matching the integer
          // value of the level and the index
          // value of the palette.
          g.setColor(colorPalette[
                                  quantizedData]);
          //Draw a square in the output image of
          // the specified color at the specified
          // location.
          g.fillRect(col * blockSize,
                     row * blockSize,
                     blockSize,
                     blockSize);
        }//end col loop
      }//end row loop

      //If axis is true, draw red lines to form
      // an origin at the center
      if(axis){
        g.setColor(Color.RED);
        g.drawLine(0,vertCenter,2*horizCenter,
                                        vertCenter);
```

```
        g.drawLine(horizCenter,0,horizCenter,
                                    2*vertCenter);
      }//end if
    }//end paint
  }//end inner class CanvasType2surface
  //=======================================//

  //Note that this is an inner class.  This class
  // is used to construct a color scale that
  // matches the color scheme used in the class
  // named CanvasType2surface.
  class CanvasType2scale extends Canvas{
    int scaleHeight = 6 * blockSize;

    CanvasType2scale(){//constructor
      //Set the size of the Canvas based on the
      // width of the surface and the size of the
      // square used to represent each value on
      // the surface.
      setSize(dataWidth * blockSize,scaleHeight);
    }//end constructor

    //Override the paint method to draw the
    // scale.
    public void paint(Graphics g){
      Color[] colorPalette = getColorPalette();

      for(int row = 1;row < scaleHeight;row++){
        for(int col = 0;col < dataWidth;col++){

          //Get the value of the point on the
          // scale surface.
          double scaleValue =
                           255.0 * col/dataWidth;
          //See the class named
          // CanvasType2surface for an
          // explanation of this color
          // algorithm.
          int quantizedData = (int)(Math.round(
                  scaleValue*(
                    colorPalette.length-1)/255));
          g.setColor(colorPalette[
                                quantizedData]);
          g.fillRect(col * blockSize,
                     row * blockSize,
                     blockSize,
                     blockSize);
        }//end col loop
      }//end row loop
    }//end paint

  }//end inner class CanvasType2scale
  //=======================================//

}//end outer class ImgMod29
```

```
Listing 29
```

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

-end-