Chapter 9

Writing an IDL Graphics Display Program

Chapter Overview

The purpose of this chapter is more ambitious than most. Even though IDL is a programming language, it is impossible to find anywhere in the official IDL documentation *how* to write an IDL program. I don't mean to suggest there is only one right way. But anyone who has looked over the shoulders of as many IDL programmers as I have knows there is definitely a distinction between a good IDL program and one that is not so good. As someone who spends a lot of time with people who are trying to learn IDL for the first time, I see a lot of not-so-good programs.

I am convinced the problem is lack of information. Most people using IDL are, after all, scientists, not computer programmers. They are bright and they are trying to get their work done. They are not trying to write elegant computer programs.

But, still... If only a couple of simple principles were followed, their programs would be so much better and so much more useful to them. This chapter is an attempt to specify what those principles might be, while at the same time showing you how to assemble and use many of the techniques that have been discussed in this book.

The task I have set for myself is to show you how to write a reasonably complex graphics display program that you can use from the IDL command line. But I want to write this program in such a way that the output can be displayed in a resizeable graphics window, printed directly from the IDL command line, or made into a PostScript file with little or no effort. The program should use colors in an intelligent way that doesn't depend upon the visual depth or color decomposition state of the output device.And finally, it should be simple to add a graphical user interface to this program, so that it can be used by someone unfamiliar with it.

Most programmers know that the best way to learn programming is by understanding code that has been written by others. But most of the time this is a daunting task, given the general lack of documentation in code and the absence of a mentor who can explain the unfamiliar elements of the code to you. My purpose here is not just to show you how to write a program, but to explain why the program is written in this particular way. Writing programs always involves making choices. The choices we make play a pivotal role in how useful the program is to us, and how easy it is to maintain and extend the program over time. In other words, we can make both helpful and non-helpful choices. By the end of the chapter you should be well on your way to understanding the distinction.

The HistoImage Program

The program I want to write will be named *HistoImage*. It is quite simple. It will display an image with axes around it. Above the image will be a colorbar that will indicate the image values. And above the colorbar will appear a histogram plot of the image pixel values. In other words, the histogram will show how many pixels in the image correspond to a particular image value. You see an illustration of how the program will appear in Figure 94. The complete listing of the program source code can be found "HistoImage Program" on page 409. And the *histoimage.pro* file is among the program files you downloaded to use with this book.

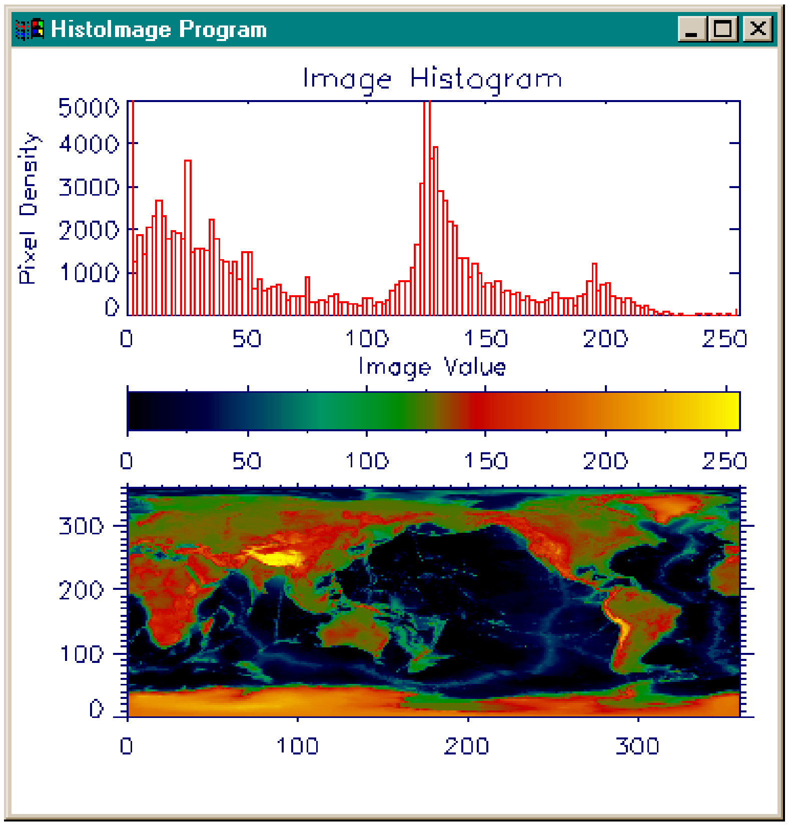

Figure 94: ***The HistoImage program consists of a histogram plot, a colorbar, and an image surrounded by axes.***

Writing the Procedure Definition Statement

The *HistoImage* program will be a procedure, so I start by defining the procedure definition statement. This is where all the positional and keyword parameters are declared. A rule of thumb for parameters is that any parameter required for the program to run will be a positional parameter, and anything else will be a keyword parameter. This means, of course, that I will have to define default values for all of my input keyword parameters.

It also means that I seldom have in mind all of the keyword parameters I am going to need when I start coding the program. Most of the time I start with a few obvious ones and add others as it occurs to me I need them. I do try to give the users of my programs as much flexibility as possible. In particular, I like to give them as much control as I can over how things are going to look, since I know users almost always have a different aesthetic sense than I do. So I will almost always have keywords that will allow the user to choose colors.

Here is what the procedure definition statement will look like:

```
PRO HistoImage, $
   image, $
   AxisColorName=axisColorName, $
   BackColorName=backcolorName, $
   Binsize=binsize, $
   ColorTable=colortable, $
   DataColorName=datacolorName, $
   Debug=debug, $
   _Extra=extra, $
   ImageColors=imagecolors, $
   Max_Value=max_value, $
   NoLoadCT=noloadct, $
   XScale=xscale, $
   YScale=yscale
```

The first and only positional parameter, *image*, is the image data that will be passed into the program. I am going to have to check whether this is a 2D array, since it doesn't make much sense to calculate the histogram of a 24-bit or 3D image, but I will delay this for a moment.

You see three "color" name keywords in the list: *AxisColorName*, *BackColorName*, and *DataColorName*. These will be the names of colors to use for the axes and other annotations, the background, and the data, respectively. I am going to use color names for these because one of my goals for the program is to use colors that are independent of the color decomposition state or depth of the display. I know that the *GetColor* program described in "Obtaining Device Independent Colors" on page 87 has the ability to give me a such a color if I ask for one of the 16 colors it knows about by name. By allowing the user to select these colors themselves, I give them some control over how things look on the display. I also allow the user to specify a color table index number with the *Colortable* keyword. The image data will be scaled into indices loaded by the color table.

Image colors almost always present a dilemma for me. On the one hand, I really want to set the image colors up correctly, especially if I am calling this program from the IDL command line. But on the other hand, I often prefer that color manipulation be done *outside* the program code. For example, in widget programs image colors are often controlled by a color table changing tool like *XLoadCT* or *XColors*. Loading a color table inside a graphics display program will often interfere with the colors that are being manipulated elsewhere.

I compromise in this program by defining two additional keyword parameters: *NoLoadCT* and *ImageColors*. *NoLoadCT* will be a flag that if set will prevent the program from loading the color table specified by the *ColorTable* keyword. In other words, I will have a way to turn color table loading off from outside the program, if this is what I choose to do. *ImageColors* will be an *output* keyword that will allow me to learn from outside the program how many colors the image data should be scaled into. This is essential information if I am to manage the colors from outside the program. (Here I will be loading image colors starting at color table index 0. If this were not the case, I might also define a keyword, say *BottomIndex*, that would indicate the bottom of the image color indices.)

The *BinSize* keyword is a *Histogram* command keyword. I will use the *Histogram* command inside this program to calculate the histogram plot and I may want the user to be able to configure the properties of this command. Note that I don't provide all of the keywords that are available for *Histogram*, only those that I specifically want the user to manipulate. Similarly, the *Max_Value* keyword is a *Plot* command keyword I often find useful in histogram plots, so I define it here.

It is entirely possible (especially with the *Plot* command) that I will want to manipulate other *Histogram* or *Plot* or *TVImage* keywords that are not defined here. For example, I may want different axes annotations or tick marks. For this reason, I include an *_Extra* keyword to take advantage of keyword inheritance, which was described in “Passing Undefined Keywords by Keyword Inheritance” on page 216.

There are two keywords, *XScale* and *YScale*, that will be used to define the range or scale of the axes that surround the image. These input keywords will be two-element arrays defining the minimum and maximum extent of the axes.

Finally, there is a *Debug* keyword, which I intend to use in the error handler portion of the code to force a traceback of where the error occurred. I typically don’t like to write error tracebacks into the command log window unless the user explicitly asks for such a thing. Using a *Debug* keyword gives me an easy way of debugging my code without frightening unfamiliar users with long lines of error text.

Writing the Error Handling Code

The next step in writing a program is to add some error handling code. Since I haven’t given a great deal of thought to the kinds of errors that might occur in the program, and since I know that users will discover errors that I didn’t anticipate anyway, I’ll write a general purpose *Catch* error handler. (The *Catch* error handler is described in “The Catch Control Statement” on page 230.) I’ll use the *Error_Message* program, which is among the programs you downloaded to use with this book and is described on page 234, because I want to take advantage of its device-independent nature. *Error_Message* uses *Dialog_Message* to report the error if it is running on an output device that supports widgets and *Message* if the device does not support widgets. *Error_Message* can also provide good error traceback information if it is required. The code looks like this:

```
Catch, theError
IF theError NE 0 THEN BEGIN
   Catch, /Cancel
   ok = Error_Message(!Error_State.Msg + ' Returning...', $
      Traceback=Keyword_Set(debug))
   RETURN
ENDIF
```

Notice that I cancel the *Catch* error handler as the first step in the error handling part of the code. I do this because I make too many typing errors during development and I sometimes have errors in my error handling code. This will cause IDL to go into an infinite loop. Call this good defensive programming if you can’t think of a gentler explanation.

Note that I am adding a bit more information (the string *Returning....*) to the normal error message. I just want the user to know what I am doing with the error. And notice that the *Traceback* keyword is set for *Error_Message* only if the user set the *Debug* keyword when he or she called the *HistoImage* program. Since the *Debug* keyword can either be on or off, I set its value with *Keyword_Set*, which only returns a 0 or 1.

Checking for Positional and Keyword Parameters

Immediately after the error handling code, goes the code that checks for all the required and optional parameters. Checking is essential (at least for input parameters) because you will be using the variables somewhere in the code to follow and it is not possible to use undefined variables in IDL expressions. (Methods for checking positional and keyword parameters are discussed in “Writing an IDL Procedure” on page 209.)

Checking for the Image Positional Parameter

I have mentioned earlier that my rule of thumb is that any required parameter is a positional parameter, and any optional parameter is a keyword parameter. I can’t do much in a program that calculates the histogram of an image and displays it above the image without an image! Hence, the *image* parameter should be a required positional argument.

But I’ve never been a person who cared much for arbitrary rules, and I’m going to choose to break this one right away by being a little kinder to the user and allowing this positional parameter to be an optional parameter. Why? Because I don’t think users should be penalized too harshly if they don’t know how to use a program. It is my job to explain it to them. So I will select and use an image for them. If they want another image, they can read the program documentation (you did write this, didn’t you?) to see how to use their own image.

The code will look like this:

```
IF N_Elements(image) EQ 0 THEN image = LoadData(7)
```

Note that I used *N_Elements* to check the image parameter, rather than *N_Params*, which you might have expected me to use for a positional parameter. Recall that *N_Elements* tells me if the image parameter is defined or not, whereas *N_Params* tells me the number of positional parameters the procedure was called with. (See “Defining Optional or Required Positional Parameters” on page 212 for more information.) Some inexperienced users are sure to call the *HistoImage* program with a single positional parameter that is an undefined variable. By using *N_Elements* I account for this eventuality.

If an *image* parameter is not supplied, I simply load the world elevation data set with the *LoadData* program you downloaded to use with this book.

Now that I have an image parameter, I will check to be sure it is a 2D array, since that is also a requirement for the histogram data to make sense. I can use the *Size* command with the *N_Dimensions* keyword set to determine the number of dimensions of the image. (Note that the *N_Dimensions* keyword to the *Size* command is a fairly recent introduction to the programming language. If you are using a version of IDL prior to IDL 5.2, you can obtain the same information from the *Size* command directly. See your on-line help for details.)

```
ndim = Size(image, /N_Dimensions)
IF ndim NE 2 THEN $
   Message, '2D Image Variable Required.', /NoName
```

The *Message* command will “throw” an error, which will be handled by my *Catch* error handling code. Note that the *NoName* keyword is set to prevent *Error_Message* from reporting the name of the program twice.

Checking for Keyword Parameters

I’m now ready to check for optional keyword parameters. I check the two keywords, *BinSize* and *Max_Value* I’m planning to use with the image histogram first. I expect most of the images used with this program will be byte data, but I can’t rely on this to be the case. Nor do I want to restrict the user to byte image data. But I do want the histogram plot of byte and, say, float image data to look the same. Thus, I am going to have 128 bins unless the user tells me something different. This will give me a reasonably good looking plot almost always. I will have to calculate the bin size appropriately for this. The code will look like this:

```
IF N_Elements(binsize) EQ 0 THEN BEGIN
   range = Max(image) - Min(image)
   binsize = 2.0 > (range / 128.0)
ENDIF
IF N_Elements(max_value) EQ 0 THEN max_value = 5000.0
```

The bin size will either be 2, or it will be the image data range divided by 128, whichever number is larger. I am making the assumption here that I am not going to have floating point image data, that ranges from 0.0 to 1.0, for example. My program will look lousy with such data, but the chances seem so low of this happening that I am willing to chance it. And, anyway, if the user did have image data like that, they could always specify an appropriate bin size for viewing the histogram.

Note the use of the IDL “greater than” operator (>). This operator returns the larger of the two values being compared. Note, too, the parentheses about the value I want to compare to the right of the operator. Without the parentheses 2.0 would be compared to the range, and then *that* value would be divided by 128. Not what I want at all! This happens because the *greater than* operator has the same order of precedence as the *division* operator. This is a common kind of error to make in IDL programs.

Another common error occurs within the parentheses. Notice I have made the number 128.0 a floating point number by adding a decimal point to the number. You might easily make the mistake of writing this number as an integer (e.g., 128). Then, for byte or integer image data, you would be dividing an integer value (the *range*) by another integer value. This might easily give you a consistent bin size of 0. A serious error.

The *Max_Value* keyword variable is assigned a value of 5000, a value I know works with most of the example image data sets distributed with IDL.

Next, I can test for the *XScale* and *YScale* keyword values. If these are not supplied, I’ll use the dimensions of the image for scale values. I’ll also test to be sure the values are two element arrays. If not, I’ll issue error messages. The code will look like this:

```
s = Size(image, /Dimensions)
IF N_Elements(xscale) EQ 0 THEN xscale = [0, s[0]]
IF N_Elements(xscale) NE 2 THEN $
   Message, 'XSCALE must be 2-element array', /NoName
IF N_Elements(yscale) EQ 0 THEN yscale = [0, s[1]]
IF N_Elements(yscale) NE 2 THEN $
   Message, 'YSCALE must be 2-element array', /NoName
```

Notice a new keyword for the *Size* command here: *Dimensions*. As opposed to the *N_Dimensions* keyword, which caused *Size* to return the number of dimensions of its argument, the *Dimensions* keyword causes *Size* to return a vector containing the size of each of its dimensions. In other words, with a 2D image array, I will get a two-element array containing the X size and Y size of the image, respectively.

Finally, I can check for the color keywords. Because I want to write device decomposed-state independent code, I am going to use the *GetColor* program to specify drawing colors. (*GetColor* is discussed in “Obtaining Device Independent Colors” on page 87.) *GetColor* “knows” the names of 16 drawing colors that I use frequently and can obtain those colors for me in a device independent way. (You can easily add more colors to *GetColor*.) I check the keywords like this:

```
IF N_Elements(dataColorName) EQ 0 THEN $
   dataColorName = "Red"
IF N_Elements(axisColorName) EQ 0 THEN $
   axisColorName = "Navy"
   IF N_Elements(backcolorName) EQ 0 THEN $
      backcolorName = "White"
```

I could check to be sure the color names passed into the program are string variables, but I know *GetColor* is going to do that anyway. And I know that it is going to return to the caller of the program that called it. Thus, I will be able to catch that error in my error handler when it comes back from *GetColor.* Thus, there is no need to check for possible errors here.

Note that I make the background color white by default. This is certainly not necessary, and more often than not I like to have a nice charcoal or gray background color. I choose white here because it is sometimes easier to visualize what the results are going to look like when I make a PostScript file from this program. Recall that you can have any background color you like in PostScript, as long as that color is white. (This problem is discussed in “Problem: PostScript Devices Use Background and Plotting Colors Differently” on page 189.)

The most important thing is that I need some way to *change* the drawing colors, because I almost certainly *will* have to change them when I send the output to a PostScript printer. By making the default colors suitable for printing on a PostScript printer now, I won’t have to worry about resetting colors. I’ll just call the *HistoImage* program to draw the graphics in its default colors when I want to make a PostScript file.

If the user doesn’t supply a color table index number, I choose color table 4.

```
   IF N_Elements(colortable) EQ 0 THEN colortable = 4
   colortable = 0 > colortable < 40
```

Note that there are only 41 color tables supplied with IDL (although users could have modified this number, certainly). Here you see a bit of checking to force the *colortable* value into a number between 0 and 40. The IDL *greater than* and *less than* operators are used in a left to right fashion to force the value into the correct range of numbers. (In other words, 0 is compared to *colortable* and the largest value is return. Then that value is compared to 40 and the smallest of those two values is returned into the *colortable* variable.) This kind of proactive error checking can avoid problems later on.

Next, I’ll supply the *ImageColors* keyword with a value. Note that I am going to use three drawing colors in this program. I prefer to load my drawing colors at the top of the color table, although other people prefer to load them at the bottom of the color table. It doesn’t matter much where you load them, as long as you know what you are doing when you manipulate the color table. But while I like to load drawing colors at the top of the color table, I don’t like to use the top index number of the color table.

The reason I don’t is that very often this index is used for the *!P.Colo*r system variable. And many, many programs are written assuming this color is going to be either white or black. I don’t like to break these programs, if I can help it. So I leave this index alone. I load my three drawing colors starting from the fourth index from the top. This means that the color indices I have for the image display ranges from 0 to the fifth index from the top of the color table. This is *!D.Table_Size* -4 total colors. This is what I assign to the *ImageColors* keyword value in the program.

```
   imagecolors = !D.Table_Size-4
```

Note that I don’t have to check this output keyword. If the user wants the value he or she can get it back from the keyword. If they don’t want the value, fine. It didn’t cost me much of anything to assign it to a variable. (And I’ll need the value later in the program anyway.) There is no need to use something like *Arg_Present* in this case:

```
   IF Arg_Present(imagecolors) THEN $
```

```
      imagecolors = !D.Table_Size-4
```

This is overkill and results in the *imagecolors* variable only being defined if the user passed in a variable reference with the keyword. The simpler construction is much easier to type and I think makes the program easier to read, too.

Remember that the *imagecolors* variable is designed to help someone outside the program control the image colors. I don't have any need for that right now, but I may later and I want to be prepared for the eventuality.

Loading the Program Colors

The three drawing colors are going to be loaded starting at the fourth index from the top of the color table. I always use *!D.Table_Size* to indicate the size of the color table. Then *!D.Table_Size-1* is the top index in the color table. The code looks like this:

```
   axisColor = GetColor(axisColorName, !D.Table_Size-2)
   dataColor = GetColor(dataColorName, !D.Table_Size-3)
   backColor = GetColor(backcolorName, !D.Table_Size-4)
```

The variables *backColor*, *dataColor*, and *axisColor* now contain either the correct color index number for loading the proper color (if device decomposition is off or if this is an 8-bit device), or a 24-bit value that can be decomposed into the proper color (if device decomposition is on). In any case, I don't have to worry at all about the current color decomposition state when I draw the graphics with these colors. The correct colors will appear almost automatically.

Next I load the colors for the image data. I only do this if the *NoLoadCT* keyword variable is *not* set.

```
   IF NOT Keyword_Set(noloadct) THEN $
      LoadCT, colortable, NColors=imagecolors, /Silent
```

Note that I use the *Silent* keyword to the *LoadCT* command. I really don't like the informational message *LoadCT* prints in the command log window every time it loads a color table. This keyword suppresses the message.

Note, too, that I restrict, with the *NColors* keyword, the number of colors loaded to just those indices used by the image. I want to be careful not to overwrite the drawing colors I just loaded. Only color table indices 0 through *!D.Table_Size*-5 will be loaded by this command.

Preparing to Draw the Graphics

The next step in writing the *HistoImage* program is to prepare to draw the graphics. I have three separate items to draw, the histogram plot, the color bar, and the image itself.

Calculating Graphic Positions in the Window

So the first thing I do is calculate the positions in the window where I want these items to go. These positions will be four-element arrays containing normalized window coordinates of the type that can be used with the *Position* keyword on most graphics commands. (See "Positioning Graphic Output in the Display Window" on page 44 for details of how this is done.) Normalized coordinates are used so the graphics will go into a window of any size. The code will look like this:

```
   histoPos =    [0.15, 0.675, 0.95, 0.95]
   colorbarPos = [0.15, 0.500, 0.95, 0.55]
   imagePos =    [0.15, 0.100, 0.95, 0.40]
```

Changing Character Size According To Window Size

One of the requirements of this program is that it can display its graphics in resizeable graphics windows. Or, put another way, that it can display graphics in a window of any size. In other words, if the window is big the histogram plot should be big and the image display should be equally big. If the window is small, the plot and image display should be small, etc. This is accomplished, obviously, by the position of the various components in the window in normalized coordinates, as described above.

But often this thinking is not carried over to the annotation of the graphic displays as well. Most programmers will use a character size of 1 and call it good. But I would like to see a big character size used in big windows and a small character size used in small windows.

There is a *CharSize* keyword that can be used with graphics commands to change the character size, but how can this be done in a way that is consistent with the size of the window? The answer, like many of the answers you have discovered so far, is to express the character size in normalized units. Unfortunately, this is impossible in IDL. Character size is *always* expressed in character units. But even so...there must be a way!

Actually, there is. It turns out that by using the *XYOutS* command you can get the *width* of a text string in normalized units. And if you use a *negative* value with the *CharSize* keyword, then *XYOutS* doesn't actually write the string to the display window, it just calculates the width of the string. For example, suppose you wanted to know the width of the text string “A Sample String”. You could type this:

```
IDL> thisSize = -1
IDL> XYOutS, 0.5, 0.5, 'A Sample String', /Normal, $
        Width=thisWidth, Charsize=thisSize
```

Now, suppose you compared the value of *thisWidth* with some target width. Say, for example, that the target width was 0.25. Another way of saying this is that the string should be wide enough to extend across 25 percent of the display window. If the target width was larger than the value of *thisWidth* you could increase the character size by some small amount, and test it again, and so on until the character size was appropriate to give you the text width you wanted. The code might look like this:

```
IDL> IF thisWidth LT 0.25 THEN thisSize = thisSize + 0.01
IDL> XYOutS, 0.5, 0.5, 'A Sample String', /Normal, $
        Width=thisWidth, Charsize=thisSize
```

Eventually, *thisWidth* would be within some small delta value of the target width. A similar algorithm can be employed if *thisWidth* is larger than the target width.

This kind of algorithm has already been developed for you in the form of the program *Str_Size* that you downloaded with the program files to use in this book. The parameters for *Str_Size* are a string and a target width, in normalized coordinates. I've found that the string “A Sample String” and a target width of 0.20 produces nicely sized characters on most plots in the graphics windows I typically use.

```
IDL> thisSize = Str_Size('A Sample String', 0.20)
```

This is a character size of approximately 1.4 for a normal sized window on a Windows machine, for example.

In this program, I will write the font character size selection code like this:

```
thisCharsize = Str_Size('A Sample String', 0.20)
```

Calculating the Image Histogram

The next step is to calculate the image histogram. (Recall that a histogram is simply a count of the number of entities in each bin of the histogram. Normally, we take this to mean the number of pixels with each image value.) I simply use the *Histogram* command, passing it the bin size I want to use, like this:

```
histdata = Histogram(image, Binsize=binsize, $
   Min=Min(image), Max=Max(image))
```

Notice I set the *Min* and *Max* keywords explicitly for the *Histogram* function to the minimum and maximum value of the image. The IDL documentation claims this is what the *Histogram* function does by default, but I have not found this to be the case. Rather, I find that it assumes a minimum value of 0 and a maximum value of 255 for byte images, no matter what the actual data values in the image.

Drawing the Graphics

There are three graphical elements I want to draw: a histogram plot, a color bar for indicating image values, and the image itself.

Drawing the Histogram Plot

IDL gives me several choices for drawing the histogram plot, but I find I don't like the two most obvious ones. Let me explain what I mean with a simple example you can type at the IDL command line. Suppose I have five bins of data, each bin being 5 units in size, and a vector that tells me how many "items" are in each bin. The *data* and *bins* vectors could be created like this:

```
IDL> data = [ 4, 6,  3,  8,  2 ]
IDL> bins = [ 0, 5, 10, 15, 20 ]
```

The most obvious approach is to simply plot the data with the *Plot* command, like this:

```
IDL> Plot, bins, data, YRange=[0,10]
```

The result, which doesn't look much like our definition of a "histogram" plot, is illustrated in Figure 95.

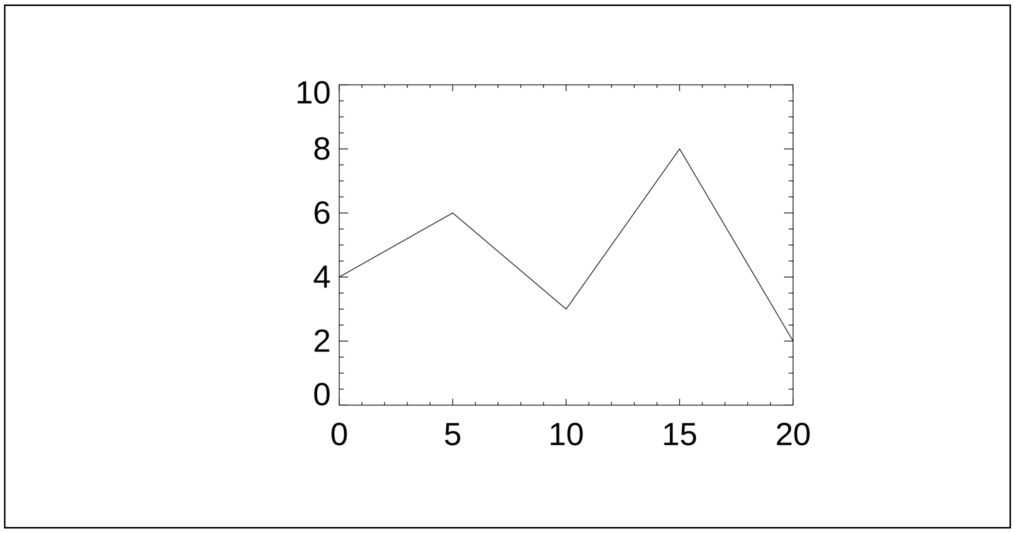

Figure 95: A very simple plot of a histogram function. Note that it doesn't look much like a histogram plot.

The second obvious approach is to set the *PSym* keyword to 10, which will result in the “stair-step” kind of plot we expect from a histogram plot. I can try this:

```
IDL> Plot, bins, data, YRange=[0,10], PSym=10
```

The results look better, as shown in Figure 96, but they are still not right. In particular, the bins are represented incorrectly. The first bin goes from 0 to 5, the second bin from 5 to 10, and so on. But in the illustration, the first bin appears to go from 0 to 2.5, and the second bin appears to go from 2.5 to 7.5, and so on. It appears as though each bin is half a bin size off.

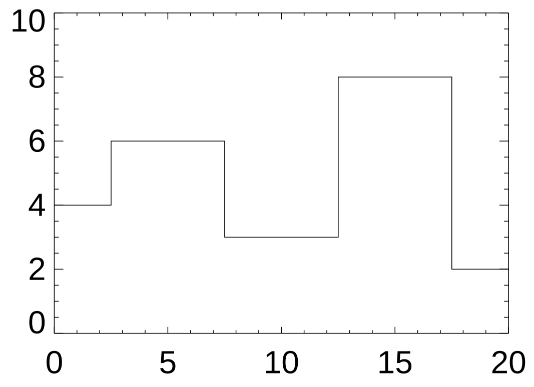

Figure 96: This plot looks more like a histogram plot, but it is still not right. Notice that the bins seem to be half a bin size off.

I could try to fix the problem by adding a half bin size to each of the bins, like this:

```
IDL> Plot, bins + 2.5, data, YRange=[0,10], PSym=10
```

You see the results in Figure 97. Again, this is close to being correct, but I have problems at either end of the plot, where the lines should extend to the end of the plot window. To really draw this plot correctly, I should duplicate the first and last values of the histogram data and the bin values.

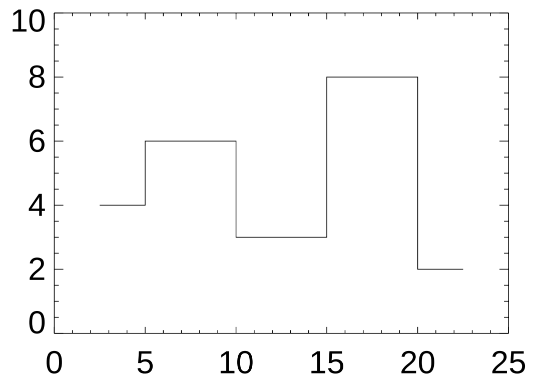

Figure 97: This plot has had half a bin size added to the bin values. It is accurate, but the plot lines do not extend to the ends of the plot window.

For example, I can do something like this:

```
IDL> Plot, [bins[0], bins + 2.5, bins[4] + 2.5 * 2], $
        [data[0], data, data[4]], YRange=[0,10], PSym=10
```

Finally, I get the kind of histogram plot I expect, as illustrated in Figure 98.

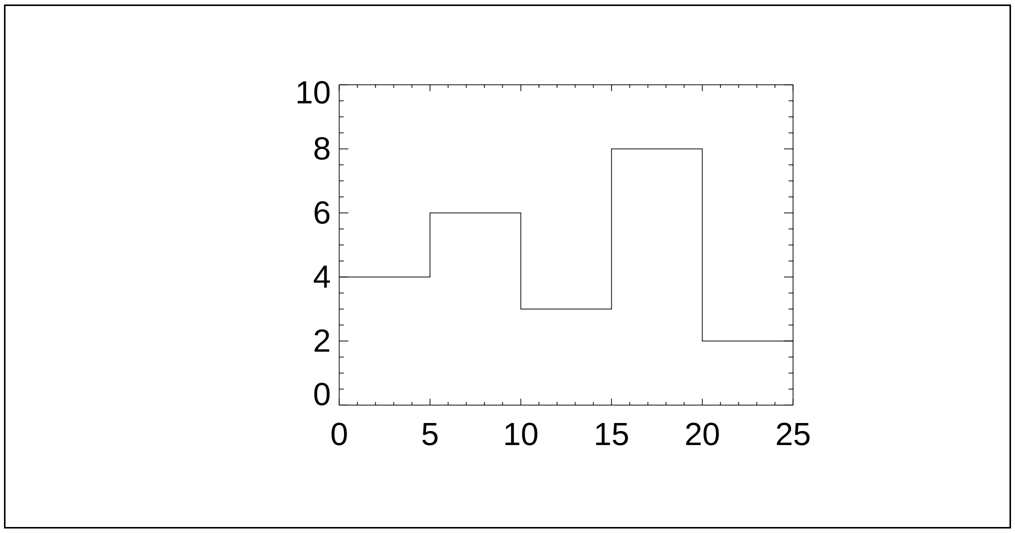

Figure 98: ***The histogram plot I expected to get. It would be easy enough to add vertical lines with the PlotS command to create histogram boxes for each bin.***

This is exactly the kind of approach I am going to take in the *HistoImage* program. The code to fudge the *bins* and *histodata* vectors for correct plotting looks like this:

```
npts = N_Elements(histdata)
halfbinsize = binsize / 2.0
bins = Findgen(N_Elements(histdata)) * binsize + Min(image)
binsToPlot = [bins[0], bins + halfbinsize, $
   bins[npts-1] + binsize]
histdataToPlot = [histdata[0], histdata, histdata[npts-1]]
xrange = [Min(binsToPlot), Max(binsToPlot)]
```

The code to draw the histogram plot in the axes color, followed by the histogram data drawn in the data color will look like this:

```
Plot, binsToPlot, histdataToPlot, $
   Background=backColor, $
   Charsize=thisCharsize, $
   Color=axiscolor, $
   Max_Value=max_value, $
   NoData=1, $
   Position=histoPos, $
   Title='Image Histogram', $
   XRange=xrange, $
   XStyle=1, $
   XTickformat='(I6)', $
   XTitle='Image Value', $
   YMinor=1, $
   YRange=[0,max_value], $
   YStyle=1, $
   YTickformat='(I6)', $
   YTitle='Pixel Density', $
```

```
      _Extra=extra
   OPlot, binsToPlot, histdataToPlot, PSym=10, Color=dataColor
   FOR j=1L,N_Elements(bins)-2 DO BEGIN
      PlotS, Color=dataColor, [bins[j], bins[j]], $
         [!Y.CRange[0], histdata[j] < max_value]
   ENDFOR
```

Notice that I use the *PlotS* command to draw vertical lines at each bin boundary. This gives the histogram a box look that I like better than just using the histogram plotting symbol (*PSym*=10) with the *Plot* command.

☞ I find that by putting the keywords in alphabetical order when I use a command that requires setting a number of keywords a useful style. It makes it much easier to see which keywords I am explicitly setting. This saves time and effort if I have to add or delete a keyword later in program development.

Drawing the Color Bar

Drawing the color bar is easy. I simply use the *Colorbar* program you downloaded to use with this book. Like *TVImage* (see "An Alternative Image Display Command" on page 62), the *Colorbar* program is device decomposition independent and can be used in any IDL supported graphics device. The commands looks like this:

```
   cbarRange = [Min(binsToPlot), Max(binsToPlot)]
   Colorbar, $
      Charsize=thisCharsize, $
      Color=axisColor, $
      Divisions=0, $
      NColors=imagecolors, $
      Position=colorbarPos, $
      Range=cbarRange, $
      XTicklen=-0.2, $
      _Extra=extra
```

Notice that the color bar is restricted to the same number of colors as the image, and that the range of colors is taken from the fudged *binsToPlot* variable. By setting the *XTickLen* keyword to a negative value, I produce outward facing tick marks. Setting the *Divisions* keyword to 0 allows the program to choose annotation divisions in the same manner as the *Plot* command. This will make the *Colorbar* annotation identical to the histogram plot annotations directly above it and will reinforce the purpose of the annotations.

Drawing the Image Plot

All that is left to do is draw the image, with its axes around it. I'll use the *TVImage* command to display the image (see "An Alternative Image Display Command" on page 62) and the *Plot* command to draw the axes, using the values given by the *XScale* and *YScale* keywords as the range of the plot. The final code will look like this:

```
   TVImage, BytScl(image, Top=imagecolors-1), $
      Position=imagePos, _Extra=extra

   PLOT, xscale, yscale, $
      Charsize=thisCharsize, $
      Color=axisColor, $
      NoData=1, $
      NoErase=1, $
      Position=imagePos, $
      XStyle=1, $
      XTicklen=-0.025, $
      YStyle=1, $
      YTicklen=-0.025, $
       _Extra=extra

   END
```

Notice that I scale the image data into the number of image colors and that I position both the image and the axes about the image with the *imagePos* variable.

Working Around a Printer Device Bug

There is a small *Printer* device bug in versions of IDL up through IDL 5.3.1 (the official version at the time this is written) that will cause a problem when this code is sent directly to a PostScript printer. (See “Loading Colors in the Printer Device” on page 204 for additional information.) It turns out that when a single color is loaded into the color table (in this program that is done with the *GetColor* command) at any index at all, then that same color is used to display any image pixel having a value of 0. (This is a *strange* bug!). For example, if you type these commands, and your default printer is a PostScript printer (PCL printers appear to be unaffected), then you might see output that looks like the illustration in Figure 99.

```
IDL> thisDevice = !D.Name
IDL> Set_Plot, 'PRINTER'
IDL> LoadCT, 0, NColors=!D.Table_Size-4
IDL> HistoImage, /NoLoadCT
IDL> Device, /Close_Document
IDL> Set_Plot, thisDevice
```

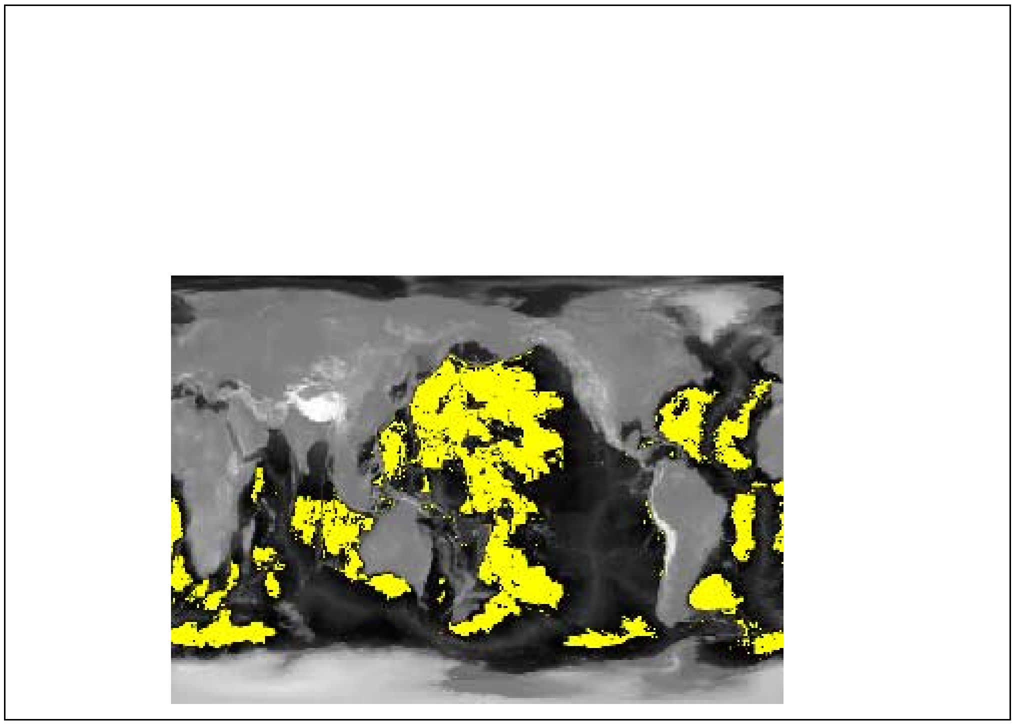

Figure 99: A Printer device bug that results in all pixels with value 0 showing up in the last single color loaded in the color table (the background color, in this case). The work-around is to get and re-load the color table vectors after loading a single color.

The work-around for this bug, which will make the code completely device independent, is to simply get and re-load the color table vectors after the last single color is loaded into the color table. In the *HistoImage* code, find this line:

```
IF NOT Keyword_Set(noloadct) THEN $
   LoadCT, colortable, NColors=imagecolors, /Silent
```

The line above should be replaced with this code:

```
IF NOT Keyword_Set(noloadct) THEN BEGIN
   LoadCT, colortable, NColors=imagecolors, /Silent
ENDIF ELSE BEGIN
   IF !D.NAME EQ 'PRINTER' THEN BEGIN
      TVLCT, r, g, b, /Get
      TVLCT, r, g, b
   ENDIF
ENDELSE
```

This will cause the correct colors to be loaded when the program is sent to the *Printer* device.

Compiling and Testing the Program

To compile and test the program, type this:

```
IDL> .Compile histoimage
IDL> HistoImage
```

If the program doesn’t compile, or if you have errors when you run it, delete the program from your display with the mouse, type *RETALL* at the IDL command line, and fix the errors. Don’t forget to re-compile the program before you try to run it again.

Try running the program with a different image:

```
IDL> image = LoadData(5)
IDL> HistoImage, image
```

Try setting some of the keywords. For example, try some of the color keywords:

```
IDL> HistoImage, image, $
        AxisColorName=’beige’, $
        BackColorName='gray', $
        ColorTable=33, $
        DataColorName='yellow'
```

Try putting different scales on the image:

```
IDL> HistoImage, XScale=[0,1], YScale=[-1,1]
```

What about setting keywords that are not defined for *HistoImage*, but will get picked up by the keyword inheritance mechanism? Try, for example, setting the *Keep_Aspect_Ratio keyword of the TVImage* command and the *Divisions* keyword of the *Colorbar* command, like this:

```
IDL> HistoImage, /Keep_Aspect_Ratio, Divisions=8
```

Reviewing the HistoImage Program’s Advantages

Let’s review some of the *HistoImage* program’s advantages. And I want to point out a few that may not be obvious to you even yet. First of all, the program has been written in such a way that it doesn’t matter whether you are on an 8-bit display or a 24-bit display, the program will work identically. Nor will it matter whether you have color decomposition on or off if you are on a 24-bit display. This is because we have used color-aware programs such as *GetColor* and *TVImage* to load drawing colors and display the image.

We have written the *HistoImage* program with an *_Extra* keyword defined for it. This allows us to pass "extra" keywords into the *Plot*, *Colorbar*, and *TVImage* commands inside the program. But it also does something far more useful. It allows us to write a program that can automatically re-display the graphic on 24-bit displays when we change color tables. (See "Automatic Updating of Graphic Displays When Color Tables are Loaded" on page 66 for additional information.)

For example, open a text editor and create this simple file, which you can name *histoimage_redisplay.pro*. (This program is one of the programs you downloaded to use with this book, if you prefer not to type it.)

```
PRO HistoImage_Redisplay, Image=image, _Extra=extra
IF N_Elements(image) EQ 0 THEN image = LoadData(7)
HistoImage, image, /NoLoadCT, _Extra=extra
END
```

This program will allow us to change the color table associated with *HistoImage* and see the effects immediately. (This will only be necessary on 24-bit displays, remember. On 8-bit displays, the colors are updated automatically.) Notice that the *NoLoadCT* keyword is set. This is necessary, you recall, for an outside entity to control the colors. If this keyword were not set, *HistoImage* would always load its own color table rather than using the colors of the current color table.

First, call the program normally and find out how many image colors there are. The *ImageColors* output keyword is used for this purpose.

```
IDL> image = LoadData(13)
IDL> HistoImage, image, ColorTable=33, ImageColors=ncolors
```

Next, call *XColors* to load different color tables. (*XColors* is a program you downloaded to use with this book. I use it exclusively in place *XLoadCT*, which is supplied with IDL, for reasons you will learn about in the following sections of this chapter. It has many advantages to *XLoadCT*, one of which is that I think it has a more natural syntax for automatically updating graphical displays on a 24-bit device.)

```
IDL> XColors, NColors=ncolors, Image=image, $
        NotifyPro='HistoImage_Redisplay'
```

If you have both *XColors* and your open graphics window on the display so you can see them both, you will notice that as you select color tables from the list of color tables in *XColors*, that the colors are automatically updated in the display window.

Note that if you are running IDL on an 8-bit device, you need only call *XColors* like this:

```
IDL> XColors, NColors=ncolors
```

Now, close your *XColors* window if it is still on your display.

The *XColors* program will work with the *HistoImage_Redisplay* program no matter what keywords you use with *HistoImage*. For example, you can type this:

```
IDL> HistoImage, image, BackColorName='gray', $
         AxisColorName='yellow', ImageColors=ncolors
IDL> XColors, Image=image, NColors=ncolors, $
        BackColorName='gray', AxisColorName='yellow', $
        NotifyPro='HistoImage_Redisplay'
```

The HistoImage Program is Device Independent

We have seen that the *HistoImage* program is visual display depth independent and color decomposition independent, but what may not be immediately obvious is that it is also device independent. That it to say, it doesn't matter which graphical display device you have currently selected to display graphics, the *HistoImage* program will work correctly in that device. This includes such devices as the PostScript device (*PS*), the printer device (*PRINTER*), and the Z-graphics buffer device (Z).

For example, to print this on your default printer, you can type this:

```
IDL> thisDevice = !D.Name
IDL> Set_Plot, 'PRINTER', /Copy
IDL> Device, XSize=5, YSize=5, /Inches, XOffset=1.75, $
        YOffset=3.0
IDL> HistoImage, image
IDL> Device, /Close_Document
IDL> Set_Plot, thisDevice
```

Or, to create a PostScript file, you can type this:

```
IDL> thisDevice = !D.Name
IDL> Set_Plot, 'PS'
IDL> Device, XSize=5, YSize=5, /Inches, XOffset=1.75, $
        YOffset=3.0
IDL> HistoImage, image
IDL> Device, /Close_File
IDL> Set_Plot, thisDevice
```

☞ Recall that I set the background color to be white by default just for the purpose of being able to send the graphic output to a printer or to the PostScript device. If it is not white on the display, you should change it to white before you send it to the printer or create a PostScript file. The color keywords make it easy to make this switch when we are using these types of graphics output devices.

What makes the *HistoImage* program device independent is the absence of commands that only work in a particular device. For example, you see no *Window* commands in this program, since a *Window* command is only appropriate on display devices and not, for example, in the PostScript device. The graphics have been written to go into any window that happens to be open. If the graphics device is a display device (i.e., *WIN*, *MAC* or *X*), and a window is not currently open, a default window will be opened when the first graphics command is executed.

If you did want to have a *Window* command in the program (and I almost never do, especially if I have plans to use the program in a widget program), then you should make sure the device supports windows. This can be done with the *Flags* field of the *!D* system variable. The flags field is a bit map and we want to see if the bit corresponding the value 256 is set (windows are supported by this device) or not (windows are not supported by this device). The code will look like this for a possible *Window* command:

```
IF (!D.Flags AND 256) NE 0 THEN Window
```

Another common command you don't see in this program is a *Device, Decomposed=0* command. Normally this command is required to display 2D images in color. We don't have to include it because the *Colorbar* and *TVImage* commands have been written with this intelligence built into them. However, we would have to use a command like this if we were going to use the *TV* command to display the color bar and image. In fact, we would probably have to use *several* commands to check the visual depth, the type of device, etc. For details, examine the code in either the *TVImage* or *Colorbar* programs. In the *Colorbar* program 12 lines of code precede the

TV command and another 12 lines follow it, just to make the *TV* command work properly on every device!

Using HistoImage in a “Smart” Resizeable Graphics Window

The *HistoImage* program also meets the graphics display criteria for being displayed with a resizeable graphics window program, named *FSC_Window*, which you downloaded to be used with this book. The *FSC_Window* program is a “smart” graphics window, in that it can resize its contents, create BMP, GIF, JPEG, PICT, PNG, TIFF, and PostScript files of its window contents, and send its contents directly to the printer. Plus, if your graphic display program has been written correctly, you can also change the colors in your program with a color table changing tool.

The five criteria *FSC_Window* imposes on a display program are these:

- 1. The program should be written as a procedure.
- 2. There should be no more than three positional arguments.
- 3. There can be an unlimited number of keyword arguments.
- 4. The program should be written so that the contents goes into any sized window.
- 5. There should be no device-specific commands in the program (e.g, a *Window* command).

Many graphics display commands meet this criteria. For example, the *Shade_Surf* command does. Type these commands:

```
IDL> peak = LoadData(2)
IDL> LoadCT, 22
IDL> FSC_Window, 'Shade_Surf', peak, Charsize=1.5
```

You see an example of the *FSC_Window* program in Figure 100. Notice the controls under the *File* button in the menu bar. You can grab the edge of the window with the mouse and resize the window. The window contents resize themselves to fit.

You can have as many *FSC_Window* programs running as you like. For example, let’s display an image in an *FSC_Window* program, but let’s also set the ability of the program to load different color tables. Type this:

```
IDL> image = BytScl(LoadData(7), Top=!D.Table_Size-1)
IDL> FSC_Window, 'TVImage', image, /WColors
```

Notice now there is a *Colors* menu item under the *File* menu bar button. Clicking this button will call up the *XColors* color changing tool. *XColors* is a program you downloaded to use with this book. One of its huge advantages is that it doesn’t use common blocks to store its color tables. Which means there can be multiple *XColors* programs on the display at once. Something that is impossible for *XLoadCT*, the IDL-supplied color table changing tool.

For example, let’s get the *HistoImage* program on the display too. Recall, though, that you do not want to use all of the colors in the color table. We reserved the top four colors for other things. And recall that if we want colors to be manipulated externally, we have to be sure *not* to load the color table in *HistoImage*. This is accomplished by setting the *NoLoadCT* keyword. Type this:

```
IDL> image = BytScl(LoadData(5), Top=!D.Table_Size-1)
IDL> FSC_Window, 'HistoImage', image, /NoLoadCT, $
        WColors=!D.Table_Size-4
```

Now you can get the color table tool from both programs on the display simultaneously. If you are running IDL on an 8-bit display, things look a bit chaotic, no doubt, since every time the color table is changed, all graphics output changes automatically.

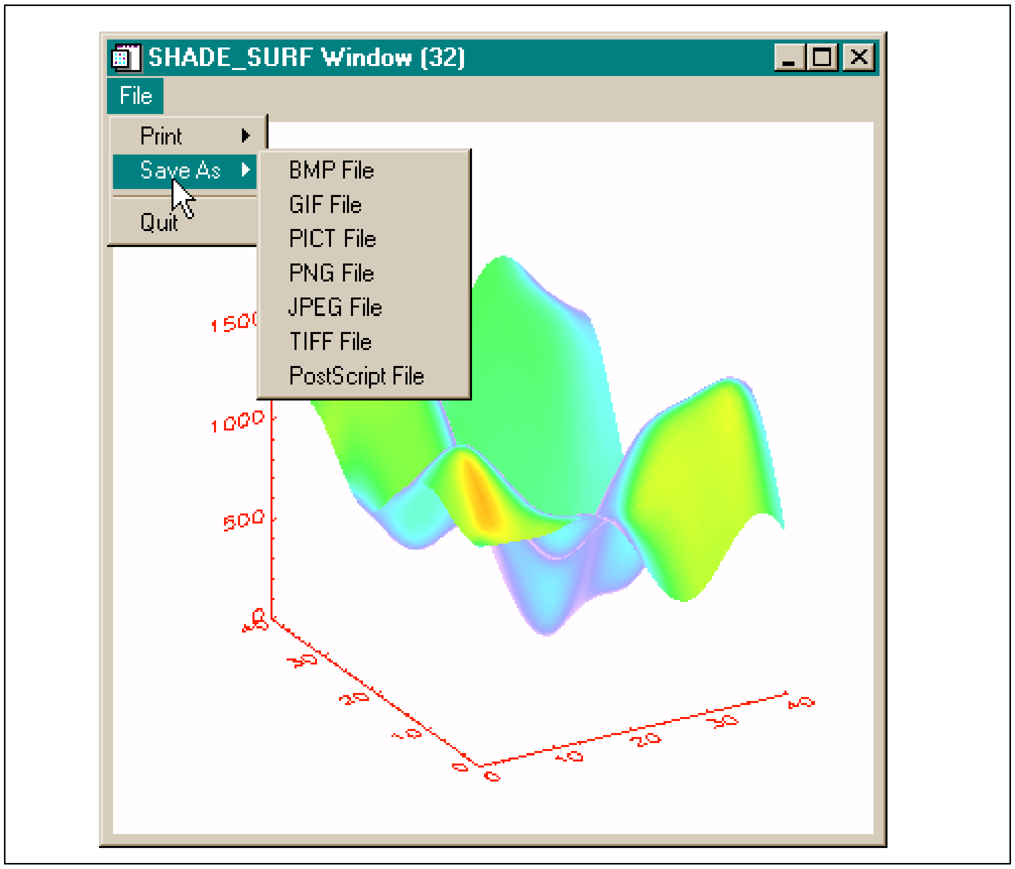

Figure 100:** **The FSC_Window resizeable graphics window program with a Shade_Surf command. Notice the pull-down menu that allows you to print and save the window contents in a variety of file formats.

On a 24-bit displays, of course, this all happens independently. What you will notice on 8-bit displays, however, is that the color will be correct for the window that has the current keyboard focus. In other words, each window "knows" which colors are supposed to be loaded and loads them when it has the focus.

☞ Note that on an 8-bit display, the *FSC_Window* program may not have a *Print* and *PostScript File* button as shown in Figure 100. When the display device does not have as many colors as the PostScript or printer devices, it is impossible to always get the colors correct for PostScript and printer output. Too many factors are involved and it depends too much on how the graphics command that is executed is written. For example, it is not possible to get correct PostScript output from our *HistoImage* program when it is running on an 8-bit display with color table loading turned off, as is the case currently.

You will learn more about how to fix these kinds of problems in the chapters to follow, but for now you should know that you can get both a *Print* button and a *PostScript File* button on your *FSC_Window* program on an 8-bit display, but you have to set them explicitly. For example, you could do this:

```
IDL> FSC_Window, 'TVImage', LoadData(7), /WColors, $
        /WPostScript, /WPrint
```

But even this command would not print correctly or make a correct PostScript file on an 8-bit display. To make the output correct, the image data will have to be scaled correctly for both the display and for the PostScript and printer devices. The only way this can be accomplished it to perform the scaling right in the *TVImage* command, like this:

```
IDL> image = LoadData(7)
IDL> FSC_Window, 'TVImage', $
        BytScl(image, Top=!D.Table_Size-1), /WColors, $
        /WPostScript, /WPrint
```