# Chapter 3


# Working with Color

## Chapter Overview

The purpose of this chapter is to present a brief introduction to color in the direct graphics system in IDL. (Color is handled differently in the object graphics system, but this will be discussed in conjunction with object graphics programming in the second half of this book.) According to participants in my IDL programming classes, nothing is as consistently frustrating as trying to get color output to work correctly on the different graphics output devices IDL supports. This chapter will explain how color works in IDL and will introduce you to several color tools that will make it easier for you to write IDL programs that work in a device-independent way.

When I wrote the 2nd Edition of this book, we were in a period of transition from 8-bit to 24-bit graphics cards. It was extremely important to know then how to write IDL programs that could co-exist in both of these environments. But I haven't run into an 8-bit graphics display in a very long time, so I presume all of us have made that transition to 24-bit graphics cards successfully. Unfortunately, we haven't been able to escape the 8-bit environment or mindset completely. We still have graphics devices (e.g., the PostScript device, the Z-graphics buffer, and most PRINTER devices) that are 8-bit devices. And an enormous amount of the IDL software we use was written with an 8-bit worldview, and that is a problem on 24-bit graphics displays. So we still have problems to overcome. This chapter is designed to help you meet those problems head on, and to help you write more flexible direct graphics programs in IDL.

Specifically, you will learn:

- How to configure a UNIX computer to work with color in IDL
- The difference between decomposed and indexed color models
- How to display color graphics in a color model independent way
- How to create, modify, and save color tables

## For Proper UNIX Colors, Start Correctly

Although never a problem for users running on Windows computers, UNIX users running versions of IDL prior to IDL 6.2 were put, by default, into an X windows color environment, *DirectColor*, from which it was next to impossible to recover. Nothing you could do in that environment made much sense. Windows were always flashing colors at you when you switched from one window to another. And many an

IDL programmer assumed that the red on black color scheme in IDL graphics windows was just the (strange) way IDL's designers had planned it.

Most of the people using IDL this way did not realize they had missed the color memo until they observed other IDL programmers working in completely different ways. Unfortunately, the memo (if there *was* one) was easy to miss. And getting their machines set up correctly was more like mystical incantations, with its strange vocabulary ("backing store," "X window visual class"), then it was like using a piece of modern software with (supposedly) helpful defaults.

You might be one of these unfortunate users. How would you know? Here is how. Start an IDL session and type the following commands. You will use the *Device* command to get information about your current graphics device.

```
IDL> Window
IDL> Device, Get_Visual_Name=theVisual, $
       Get_Visual_Depth=theDepth
IDL> Print, theVisual, theDepth
```

If the visual name is *DirectColor*, then you can be completely excused for not understanding how color works in IDL. No one else does, either, in that visual environment.

I think almost everyone these days will see the depth as 24, meaning a 24-bit graphics display, which is typical. (A depth of 16 would be treated as a 24-bit depth, for most of this color discussion, so that is okay, too.) If you have an 8-bit depth or a *PseudoColor* visual name, then you are stuck with an ancient computer and we feel your pain, but you are probably in good shape for colors. If you are reading this book, it is probably because a few of your colleagues with newer computers are complaining about your programs and would like you to know how to write programs that can co-exist in both environments. The discussion that follows will certainly help.

## Be Sure You Are In a TrueColor Visual Class

If you have a 24-bit graphics card (the depth was 24 in the commands above), then you want to be using a *TrueColor* visual class, *not* a *DirectColor* visual class. (If you have an 8-bit depth, then you should be using the *PseudoColor* visual class.) Unfortunately, the visual class is selected at the moment when IDL opens its first graphics window, and cannot be changed in that IDL session.

Selection of an X windows visual class can be done in one of two places. You can modify your *.Xdefaults* file to include the *idl.gr_visual* and *idl.gr_depth* resources, like this:

```
idl.gr_visual: TrueColor
idl.gr_depth: 24
```

Or, you can modify your IDL startup file to select a 24-bit *TrueColor* visual (see "Using an IDL Startup File" on page 15) by adding the following command to your IDL startup file.

```
Device, True_Color=24
```

You will have to exit IDL and restart it for these changes to take effect.

Now you will at least have put yourself in a position from which colors *can* be understood, although it will probably still require diligent study. (As a warning, I should point out that even those of us who consider ourselves reasonably knowledgeable in the color arena find ourselves scratching our heads a great deal more frequently than seems absolutely necessary. Your mileage may vary, too.) But, read on.

# Understanding IDL Color Models

The central problem to be overcome in trying to understand color in IDL is this: two completely different color models can be used to specify colors on a 24-bit graphics display, and the same IDL direct graphics commands work differently depending upon which model is currently in use. And the problem is only compounded by the fact that some IDL commands (those written by many of your colleagues, no doubt, but some common ones in IDL, too) will only work in one model and not the other. And then, of course, colors work differently on Windows and UNIX machines. Sigh... It does takes some time and experience to sort it all out.

The two color models are called *decomposed color* and *indexed color*. We often refer to these models as "decomposition on" and "decomposition off," respectively, because of the way each model is selected with the *Device* command. The *Decomposed* keyword is set to 1 to indicate the *decomposed color model*, and is set to 0 to indicate the *indexed color model*.

```
Device, Decomposed=1 ; Selects the Decomposed Color Model.
Device, Decomposed=0 ; Selects the Indexed Color Model.
```

By default, when IDL starts up in a 24-bit *TrueColor* environment, it will be using the decomposed color model. Or, another way to say this, color decomposition is *on*. But, what does this mean?

Every color in IDL is represented, ultimately, as a three-element byte vector of red, green, and blue values, in which each value can vary between 0 and 255. We call this a *color triple*. Thus, we have the possibility of specifying (256 times 256 times 256) approximately 16.7 million possible colors in IDL. We say we have a palette of 16.7 million colors to choose from. This is also known as *true color*, because it is similar enough to what we see with our eyes in the real world to be a reasonable representation of that world.

Our two color models arise from how we select a color from this color palette. We might wish to select one of the 16.7 million colors directly, by specifying its color triple, or we might wish to load a specific 256 colors out of our 16.7 million color palette colors into a color lookup table (which can only be 256 elements in length) and specify a color by means of its index into that color table. If we decide to select our color directly, we must specify a color triple. But, rather than using a three-element vector, as is done in the object graphics system in IDL, in the direct graphics system we create a 24-bit value that can be *decomposed* into three 8-bit values. This is what is meant by *color decomposition*.

Consider a yellow color, which is the color triple [255, 255, 0]. (The first element is red, the second green, and the third blue.) To construct a 24 bit value that can be decomposed into this color triple, we write code like this:

```
IDL> color = [255, 255, 0]
IDL> thisColor_d = color[0] + color[1]*2L^8 + color[2]*2L^16
IDL> Print, thisColor
     65535
```

Note that the lowest 8 bits in this 24-bit value represent red bits, the next 8 bits represent green bits, and the next 8 bits represents blue bits. (The highest 8 bits in this 32-bit long integer value are not set and are all zeros.) Displayed as a binary value, with the highest 8 bits removed, the number looks like this, with the lowest 8 bits on the right:

```
0 0 0 0 0 0 0 0  1 1 1 1 1 1 1 1  1 1 1 1 1 1 1 1
```

If we wanted to express this yellow color as a color index, we would have to load the color into the color table at a particular color index. Suppose we load it at color index 200 with the *TVLCT* command, like this:

```
IDL> TVLCT, 255, 255, 0, 200
IDL> thisColor = 200
```

The last argument to *TVLCT* is the color index where we are loading the color triple [255, 255, 0]. This is how we will access this color when using the indexed color model. Note that a single index (0 to 255) is used to select three separate values: the red, green, and blue color values associated with that index in the color table.

We see now that the same yellow color can be represented as a 24-bit number (*thisColor = 65535*) or as a color index (*thisColor = 200*) in IDL. In the vast majority of IDL graphics output commands colors are input as a value to a *Color* keyword (or an equivalent keyword like *Background*, etc.). Whether that value is interpreted as a value to be decomposed or as an index number into a color table depends on what color model is currently selected in the IDL session. We say it depends on the *color decomposition state* of the IDL session.

Naturally, you can get strange results if the color value you supply is mismatched with the color model that can interpret the value appropriately. Most IDL users run into problems when they use color values that represent color index numbers in their code, but they use the decomposed color model (which, remember, is the default color model) that interprets those color values as numbers to be decomposed. If you decompose any number from 0 to 255 (which are valid color table index numbers) into a 24-bit value, the only bits you can possibly set are those bits used to represent red colors. For example, the binary value 200 is represented like this:

```
0 0 0 0 0 0 0 0  0 0 0 0 0 0 0 0  1 1 0 0 1 0 0 0
```

Now, do you understand why you might be seeing red plots on black backgrounds in IDL? Here is an example of exactly this sort of mismatch between a color value and a color model.

```
IDL> Device, Decomposed=1
IDL> Plot, findgen(11), Color=200
```

The solution, of course, is to match your color model with the color representation of your number (an indexed color model, for example, when we use the color index number 200 to represent a yellow color).

```
IDL> Device, Decomposed=0
IDL> Plot, Findgen(11), Color=200
```

Or, done the other way around.

```
IDL> Erase
IDL> Device, Decomposed=1
IDL> Plot, Findgen(11), Color=65535L
```

Because so much software has been written in the past with an 8-bit worldview (as IDL itself was, several years ago), many users find it advantageous to make sure they use the indexed color model, which necessarily limits them to 256 colors. So you will see the following line in a lot of IDL startup files (see “Using an IDL Startup File” on page 15):

```
Device, Decomposed=0
```

Although not *exactly* the same as having to dress in mustard yellow shirts with wide, paisley ties and bell-bottomed pants to go to work, it does tend to date you, nonetheless. (I’m looking at a mid-1970s family wedding photo for a reference here.

I'm not even going to mention mustaches and mutton-chop sideburns. Who *were* those people!?)

No one we want to resemble, I hope. So I have taken it on as my special mission to the IDL community to teach people to take advantage of that 24-bit graphics card they paid so much money for and learn to specify the 16.7 million colors in a better way.

## Specifying Colors in a Device Independent Way

Here is the problem, as I see it. The kind of code we are talking about writing, doesn't exactly give me the warm, cozy feeling of "yellow."

```
Plot, Findgen(11), Color=200
Plot, Findgen(11), Color=65535L
```

The whole "color as number" scenario doesn't make much sense to me. Especially when I am busy trying to figure out what color model or "decomposition state" I happen to be in when I get around to displaying some graphics. It would make a lot more sense to be able to write code like this:

```
Plot, Findgen(11), Color='yellow'
```

And expect to find a yellow plot on my display no matter what decomposition state I am in or color model I am using when I type the command.

Of course, IDL doesn't work this way. But we can't have IDL dictating how we work, or we will all go paranoid and schizophrenic, sure enough. So I have written a little "independent color" program, named *FSC_Color*. With *FSC_Color* I can write code like the following and I can always expect a yellow plot to appear on my display, no matter what color model is currently in place.

```
IDL> Plot, Findgen(11), Color=FSC_Color('yellow')
```

How does it work, and how many colors does it know about?

The program currently "knows" the names of 104 colors. I chose these from a spectrum of colors to represent various drawing colors I would like to use in my own IDL programs. But if you don't like my colors, you can load your own from a text file that you can create. You can list the 104 colors in alphabetical order, like this:

```
IDL> Print, FSC_Color(/Names)[Sort(FSC_Color(/Names))], $
        Format='(6A18)'

Active  Almond  Antique White  Aquamarine Beige  Bisque
Black  Blue Blue Violet  Brown  Burlywood  Cadet Blue
Charcoal  Chartreuse  Chocolate  Coral  Cornflower Blue
Cornsilk  Crimson  Cyan  Dark Goldenrod  Dark Gray Dark Green
Dark Khaki  Dark Orchid  Dark Red  Dark Salmon  Deep Pink
Dark Slate Blue  Dodger Blue Edge  Face  Firebrick  Frame
Highlight  Honeydew  Hot Pink  Indian Red  Ivory  Khaki
Lavender  Lawn Green  Light Coral  Light Cyan  Light Gray
Light Salmon  Light Sea Green  Light Yellow  Lime Green
Linen  Magenta  Maroon  Medium Gray  Medium Orchid
Moccasin  Navy  Olive  Olive Drab  Orange  Orange Red
Orchid  Pale Goldenrod  Pale Green  Papaya  Peru  Pink
Plum  Powder Blue  Purple  Red Rose  Rosy Brown  Royal Blue
Saddle Brown  Salmon  Sandy Brown  Sea Green  Seashell
Selected  Shadow  Sienna  Sky Blue  Slate Blue Slate Gray
Snow  Spring Green Steel Blue  Tan  Teal  Text  Thistle
Tomato  Turquoise  Violet  Violet Red  Wheat  White  Yellow
```

If you don't know the name of a color to use, *FSC_Color* allows you to select a color interactively from a palette of colors. Use the *SelectColor* keyword like this. (Note that the *Coyote* program *PickColorName* also allows you to select a color name.)

```
IDL> Plot, Findgen(11), Color=FSC_Color(/SelectName)
```

You will see something that looks like this.

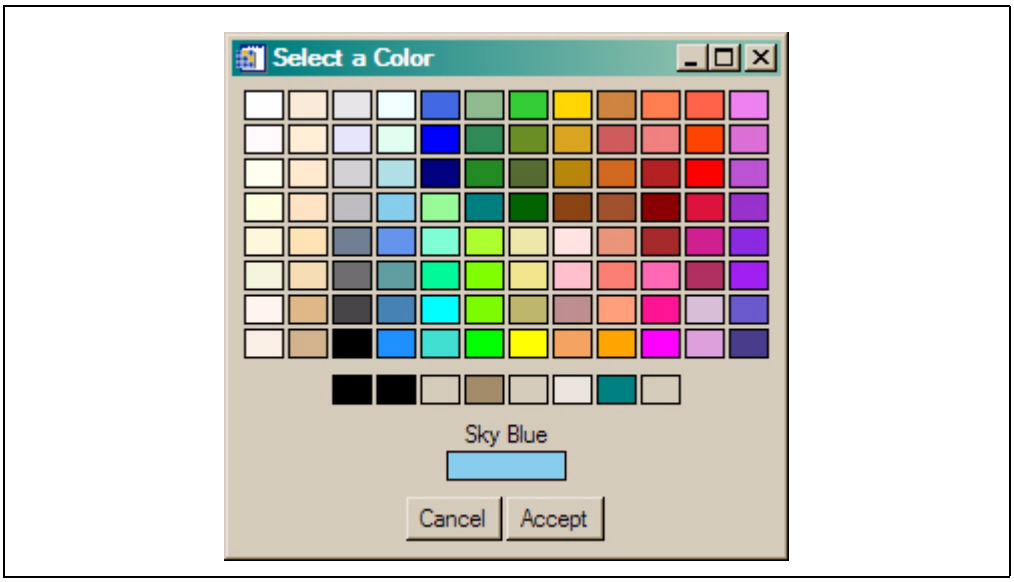


***Figure 1:*** ***The FSC_Color program will allow you to select a color interactively if you don't know the color's name. There are 104 colors available. The row of eight colors along the bottom of the palette are "system" colors associated with your operating system. You can use these colors to create graphics windows and widgets that look similar to other windows that appear on your display.***

The program works very simply. It has four vectors internally. One vector is filled with color names, the other three vectors are filled with the red, green, and blue values of the colors associated with those names. Here is a simplified representation of the four vectors.

```
names = ['teal', 'khaki', 'salmon']
r =     [    0,    240,      250 ]
g =     [  128,    230,      128 ]
b =     [  128,    140,      114 ]
```

When you ask for a color name, I look the name up in the *names* vector with the *Where* function, find it's index, and use that index to find the corresponding red, green, and blue value in the color vectors to create the color triple.

```
theIndex = Where(StrUpCase(names) EQ 'KHAKI')
colorTriple = [r[theIndex], g[theIndex], b[theIndex]]
```

Next, I determine what color decomposition state is currently in effect for this IDL session.

```
Device, Get_Decomposed=currentState
```

If color decomposition is turned on, I create a 24-bit integer value from the color triple, and return that as the result of the function. I use *Color24*, another Coyote program, to create the 24-bit value.

```
IF currentState EQ 1 THEN Return, Color24(colorTriple)
```

If, however, color decomposition is turned off, then I load the color at a particular color index number, and I return the color index number.

```
IF currentState EQ 0 THEN BEGIN
   TVLCT, Reform(colorTriple, 1, 3), 255-(theIndex)-1
   RETURN, 255-(theIndex)-1
ENDFOR
```

The 104 colors are designed to load themselves at unique indices in the top half of the color table. Under no circumstances (unless forced, see below) will they load themselves at index number 255. This makes it possible to use various drawing colors on your display and in PostScript files, for example, without having to think overly much about where those colors should be loaded in a color table. That is to say, this method *generally* does the right thing.

But there are times when you wish a color to be loaded at a particular color index number. You can do that with *FSC_Color* by simply specifying what that index number should be. For example, if you wish to load a yellow color at color index 240, you can call *FSC_Color* like this:

```
color = FSC_Color('yellow', 240)
```

Note that the value in the variable *color* will depend on the decomposition state in effect when this command is issued. Colors are actually loaded into the color table *only* if color decomposition is turned off. Otherwise, colors are turned into 24-bit values that can be decomposed into the proper color values.

```
IDL> Device, Decomposed=1
IDL> Print, FSC_Color('yellow')
     65535
IDL> Print, FSC_Color('yellow', 240)
     65535
IDL> Device, Decomposed=0
IDL> Print, FSC_Color('yellow')
     205
IDL> Print, FSC_Color('yellow', 240)
     240
```

While the *FSC_Color* program was originally designed to select colors at the moment graphics commands are being executed, there are times when colors have to be pre-loaded into the color table (e.g., when outputting to a PRINTER device, or when drawing a filled contour plot). The *FSC_Color* program has been modified to help with that. Setting the *Triple* keyword will result in a color triple being returned instead of the usual output. The triple is returned as a column vector, which will allow it to be used as input to the *TVLCT* command that loads colors in the current color table. So, for example, if you are pre-loading a color table and you wish to have yellow at color index 200 (regardless of the current color model), you can type code like this:

```
IDL> LoadCT, 0, NColors=200, /Silent
IDL> TVLCT, FSC_Color('yellow', /Triple), 200
```

In fact, multiple colors can be loaded by specifying a vector of color names, rather than a single color name.

```
IDL> TVLCT, FSC_Color(['teal', 'khaki', 'salmon'], /Triple),
        201
```

You can see what colors you currently have loaded in your color table by using the *CIndex* program (another *Coyote* program). You will see the yellow, teal, khaki, and salmon colors loaded at color indices 200 to 204.

```
IDL> CIndex
```

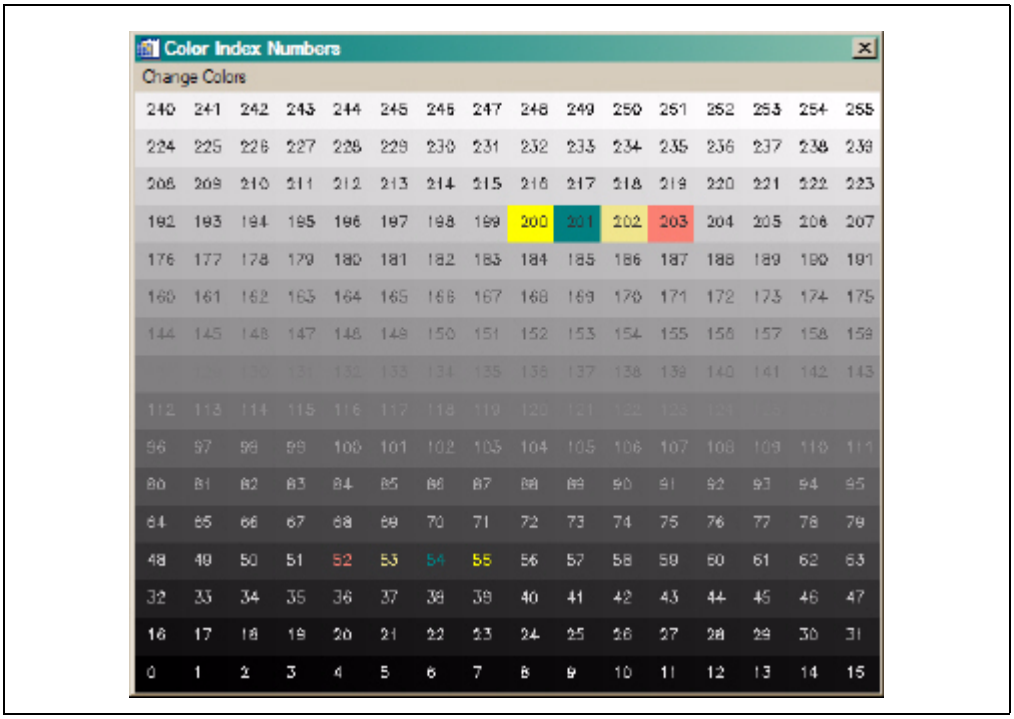


***Figure 2:*** ***The Coyote program CIndex will show you the colors currently loaded in your color table. You may keep CIndex on your display as you work with color tables. To update CIndex to the current colors, you must click the cursor inside its main graphics window. Note that the colors appear on the index numbers 52 though 55, too. This is because to see the index numbers on the colors I have to write the numbers in the "opposite" color. 255 minus index 203 is 52.***

To see all the *FSC_Color* colors loaded in the color table, starting a color index 64, type the following command, then click you cursor inside the *CIndex* window to update its display.

```
IDL> TVLCT, FSC_Color(/All, /Triple), 64
```

The *FSC_Color* colors are loaded in indices 64 through 167.

You will learn a great deal more about the benefits of using a program like *FSC_Color* to specify your graphics drawing colors in the chapters that follow, as we will make extensive use of it to write color model and graphics device independent IDL programs.

## Color Models Also Affect Image Display

Probably the number one reason we see so many IDL users limiting themselves to 256 colors by selecting the indexed color model in their IDL startup files is because the choice of color model also affects the display of images with the *TV* and *TVScl* commands. In particular, if you have the decomposed color model selected (it is the default color mode, remember) and you load a color table, and display a 2D image, the image is *not* displayed in color. Enormously frustrating!

Here are some commands you can type to see what I mean.

```
IDL> Device, Decomposed=1
IDL> LoadCT, 22
IDL> image = LoadData(7)
IDL> TV, image
```


***Figure 3: When using the color decomposed model 2D images are always displayed in gray-scale colors, even when a color table is loaded.***

The image, which is suppose to be seen in nice pastel colors, is displayed instead in gray-scale colors. And it doesn't matter what color table we load, all we can get out of this situation is gray-scale colors.

To display the image correctly, we have to switch to the indexed color model.

```
IDL> Device, Decomposed=0
IDL> TV,image
```

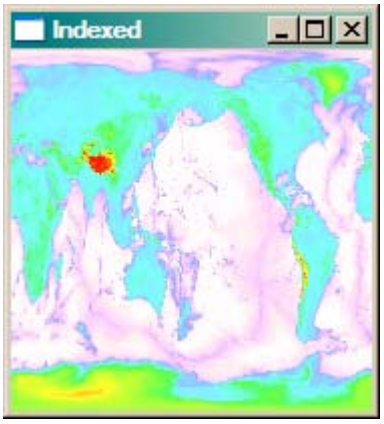


***Figure 4: To get 2D images to display in color, we have to use the indexed color model.***

What accounts for this? I'm not sure. I've always thought that IDL was "building" a 24-bit (also called a *true-color image*) image out of the 8-bit image by replicating the 8-bit image three times. Any 24-bit image of this type will necessarily be displayed in gray-scale.

## Displaying 24-bit Images

But there is also a problem in how 24-bit images are displayed, at least on machines running Microsoft Windows operating systems. Consider this 24-bit rose image.

```
IDL> rose = LoadData(16)
IDL> Help, rose
     ROSE            BYTE      = Array[3, 227, 149]
```

A 24-bit image (this one is pixel interleaved) has color information built into the image itself. It displays normally with a decomposed color model.

```
IDL> Device, Decomposed=1
IDL> TV, rose, True=1
```


***Figure 5: A 24-bit image is displayed correctly with the decomposed color model.***

But if we use the indexed color model, the image is displayed correctly on Windows machines only if the gray-scale color table is loaded. It displays incorrectly if any other color table is loaded. The image is always displayed correctly on UNIX machines. But, of course, UNIX users have to be aware of this to write machine portable IDL code.

```
IDL> Device, Decomposed=0
IDL> Window, XSize=227*2, YSize=149, Title='Indexed'
IDL> LoadCT, 0, /Silent
IDL> TV, rose, True=1, 0
IDL> LoadCT, 22, /Silent
IDL> TV, rose, True=1, 1
```


***Figure 6: A 24-bit image is only displayed correctly with the indexed color model if the gray-scale color table is loaded. This happens only on the Windows operating system, but, of course, you have to plan for it if you want to write machine-portable IDL code.***

What happens in this case is that the RGB values in the 24-bit image, which in fact represent the colors the user wants to display, are routed through the color tables to look up different RGB values for the display of the image. Yikes! Who thought this was a good idea?

For this reason, and many others (which you will learn about in more detail in the image display chapters), a great many IDL users no longer use the *TV* (or *TVScl*) command to display images. Instead, they use one of several smart *TV* substitute commands that can be found in IDL libraries on the Internet. These commands determine which color model is in use at the time the command is used, switch to the proper model to display the image correctly, then switch back to the starting color model after the image is displayed. Thus, both 8-bit and 24-bit images are always displayed in the proper color, regardless of the color model currently in effect.

The most popular of these substitute commands are *TVImage*, a *Coyote* library program, and *ImDisp*, a *TV* substitute command written by Liam Gumley and available from his web page (*http://cimss.ssec.wisc.edu/~gumley/index.html*). We make extensive use of *TVImage* (or, sometimes, *TVScale*) in this book.

For example, to have *TVImage* act exactly like the a *TV* command with more color intelligence, you must only set the *TV* keyword. Note that *TVImage* can also determine from the 24-bit image you are trying to display what kind of image interleaving is required, so there is no reason to use the *True* keyword in the call. This makes it easier to display 8-bit and 24-bit images with the same IDL code.

```
IDL> Device, Decomposed=1
IDL> Window, XSize=227*2, YSize=149, Title='TVImage Indexed'
IDL> LoadCT, 22, /Silent
IDL> TVImage, Congrid(image, 227,149), 0, /TV
IDL> Device, Decomposed=0
IDL> TVImage, rose, 1, /TV
```

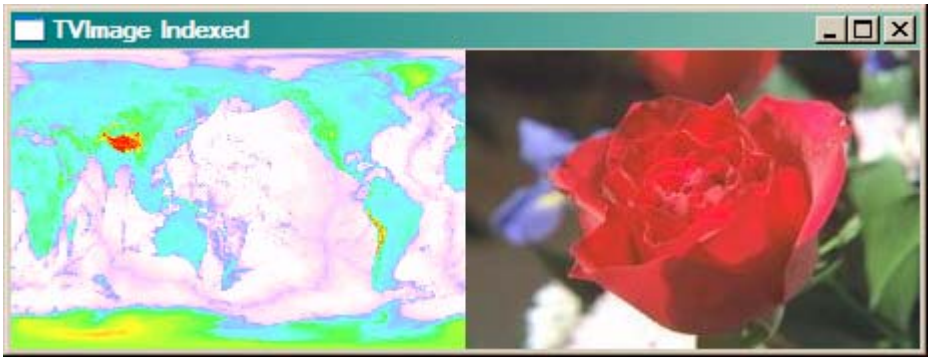


***Figure 7: The TVImage command works identically in decomposed color or indexed color models to produce images of the correct color, and doesn't require the use of the True keyword to indicate image interleaving. These are only two of the many additional advantages to using substitute TV commands.***

## Working with Color Tables

IDL comes with a standard set of 41 color tables, found in the file *colors1.tbl*, which is located in the */resource/colors/* sub-directory of the IDL distribution. The files are normally accessed and loaded by either the *LoadCT* or *XLoadCT* command. The *LoadCT* command, which you have already used in this chapter, is normally used when you know exactly which color table you want to load. For example, to load the standard gamma II color table, which is color table index 5 in the *colors1.tbl* file, you would issue a command like this.

```
LoadCT, 5
```

Using the *LoadCT* command masks, to some extent, what is really happening in IDL. Since the *LoadCT* command is an IDL library file, you could open the file in a text editor and read the IDL code to find out what it does. You would find that it reads three vectors from the color table file. We call these the red, green, and blue color vectors, and each vector contains 256 elements. Those vectors are loaded into the color table with the *TVLCT* command, which is the fundamental command for loading colors in IDL. The *TVLCT* command loads color vectors of any length from 1 to 256.

```
TVLCT, red, green, blue
```

Depending upon the size of the color table, which is always stored in the system variable *!D.Table_Size*, and is always 256 if you have a 24-bit graphics card, these color vectors are sometimes resampled before they are loaded. That is to say, if you had a color table with only 96 entries, these color vectors would be resampled to 96 colors, and those colors loaded with *TVLCT*. The resampling is done with the *Congrid* command. The resampling is a statistical process in which the end points are kept fixed, and colors (values, really) are dropped out of the larger vector in a more or less uniform manner, so that the reduced number of colors more or less represents the color range in the larger vector. So, for example, if you had 96 colors in your color table, or if you only wanted to use 96 colors in a particular color table, you could resample and load your color vectors, like this:

```
r = Congrid(red, 96)
g = Congrid(green, 96)
b = Congrid(blue, 96)
TVLCT, r, g, b
```

If you wanted to load those 96 colors, but you wanted to start loading them at color index 64, rather than zero, so that they were loaded at color indices 64 through 159, then you would use a fourth positional parameter to *TVLCT*, which is the starting color index number.

```
TVLCT, r, g, b, 64
```

As it happens, you can do the exact same thing, with the *LoadCT* command, by using the *NColors* and *Bottom* keywords. To see what I mean, start with the default gray-scale color table (color index 0), and load the Hue-Sat-Value-2 color table (color index 22) into the 96 color indices, starting at color index 64. View the results by using the *CIndex* command.

```
IDL> LoadCT, 0
IDL> LoadCT, 22, NColors=96, Bottom=64
IDL> CIndex
```

Note that the *LoadCT* command will issue a message in the command log window whenever a new color table is loaded. I have always found this more of an annoyance than a help, especially in widget programming. If you wish to turn this messaging off, use the *Silent* keyword to the *LoadCT* command.

```
IDL> LoadCT, 0, /Silent
```

## Loading Color Tables Interactively

Sometimes you do not know which color table you want to you, or you would like to try several color tables to see which conveys the most information to you from your data, or you would like the user of your program to make a color table choice of their own. In such a case, we allow the user to select a color table interactively. The command supplied with IDL to do this is *XLoadCT*.

You will have to learn to use *XLoadCT* on your own. Color is extremely important to me, and I use it extensively in the IDL programs I write. I find *XLoadCT* to be

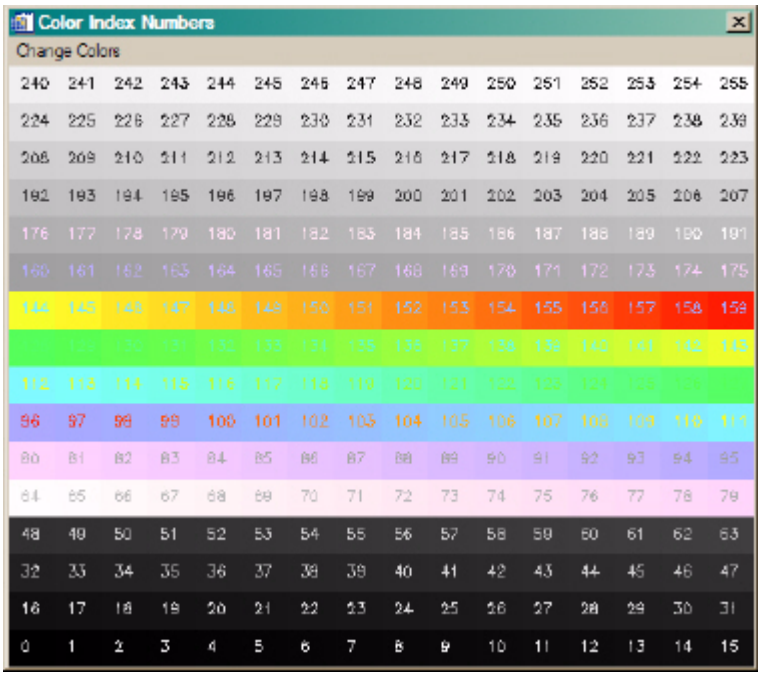


***Figure 8: The Hue-Light-Value-2 color table is resampled to 96 colors and loaded into the color table starting at color index 64 by using the NColors and Bottom keywords to the LoadCT command.***

deficient in the ways I want to use it, so I have not used it in at least the past eight or nice years. Rather, I am going to use the *Coyote* program *XColors* to load colors interactively in this book. *XColors* is written in such a way that it can be a drop-in replacement for *XLoadCT* nearly all the time. It uses the same color tables, the same keywords, etc. It just does a few things, which I seem to always require in my programs, much better than *XLoadCT.* These are primarily in the area of program to program communication. You will learn more about these advantages, including why *XColors* doesn't use common blocks and why that makes sense for a color table tool, in subsequent chapters in this book. Oh, and *XColors* works correctly when you are using a decomposed color model. That is a *big* advantage!

To see how *XColors* can be used to communicate between programs, first clear your display of any widget programs currently running.

```
IDL> Widget_Control, /Reset
```

Now call *CIndex*, but use the *NotifyID* keyword to obtain the widget identifiers of the *Change Colors* button and the top-level base widget.

```
IDL> LoadCT, 0
IDL> CIndex, NotifyID=theIDs
```

These identifiers can be passed to *XColors* through its own *NotifyID* keyword. Now, when *XColors* loads a new color table, the *CIndex* program is notified of that fact by *XColors* sending a widget event to the *CIndex* program. *CIndex* responds to the event by updating its color display. This allows you to select color tables with *XColors* and see the effect immediately in *CIndex*, a completely different widget program. To change just those 96 colors, starting at color index 64, as before, type this:

```
IDL> XColors, NotifyID=theIDs, NColors=96, Bottom=64
```

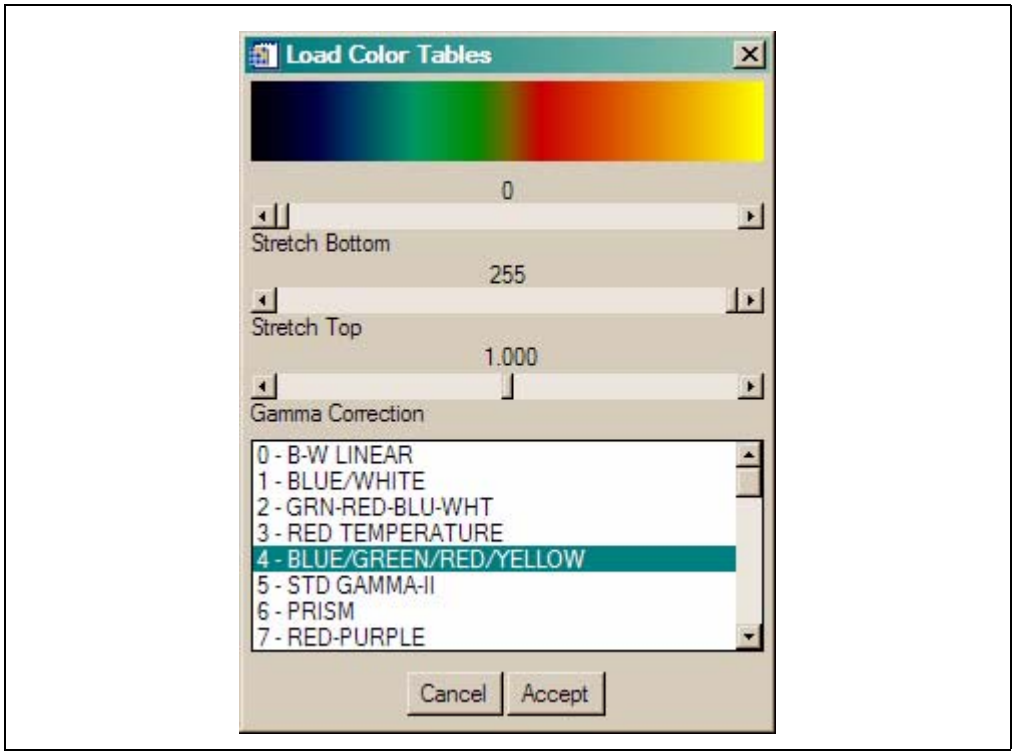


***Figure 9:*** ***The XColors program, which is used in place of XLoadCT in this book. XColors is better at program-to-program communication and avoids the use of common blocks, which makes it more versatile in IDL programs.***

Note you may have to minimize your IDLDE window to see both the *CIndex* and *XColors* windows at the same time, or you won't see the immediate update I am talking about.

A typical frustrating problem for beginning IDL programers is that they display an image in an IDL graphics window, and then they want the image colors to be updated as they change color tables with a tool like *XColors*. Normally, on a 24-bit display, to see the new colors in effect for your image, you would have to change the color table, then re-display the image to take advantage of the new color table.

But there is an easier way. You can write a short IDL program that does it automatically. Open a text editor and type the following short program. When you are finished typing, save the file as *refresh.pro* in a directory on your IDL path. (Your current directory is also a good place to save the file.)

```
PRO Refresh, IMAGE=image, WID=wid, _EXTRA=extra
   IF N_Elements(wid) NE 0 THEN WSet, wid
   TVIMAGE, image, _Extra=extra
END
```

Compile the program like this:

```
IDL> .compile refresh
```

If the program doesn't compile, fix the errors (probably typing errors if you type like everyone else around here, sigh...), save the file, and try again to compile it. Repeat until perfect.

Now, load an image and display it in a graphics window.

```
IDL> image = LoadData(7)
IDL> LoadCT, 0
```

```
IDL> Window, 1, Title='Window 1'
IDL> TVImage, image
```

To change the colors for this image, and see them updated immediately in Window 1, type the following command. Note that you may have to minimize your IDLDE window to be able to see both the Window 1 and *XColors* at the same time.

```
IDL> XColors, NotifyPro='Refresh', Image=image, WID=1, $
       Title='Window 1 Colors'
```

While those windows are still on the display, open a second graphics window, and display the image there. You can control the colors to the second graphics window with another copy of *XColors*. (This cannot be done with *XLoadCT* because it uses common blocks, and must therefore limit itself to one copy of itself on the display at any one time.)

```
IDL> Window, 2, Title='Window 2'
IDL> TVImage, image
IDL> XColors, NotifyPro='Refresh', Image=image, WID=2, $
       Title='Window 2 Colors'
```

Given the name of an IDL procedure to call, *XColors* will pass along any information passed to it in keywords it does not have defined for itself. It passed this information along anytime the color table changes. This is an extremely flexible and general method for program to program communication, and makes it quite simple and easy to write programs that work the way you want and expect them to work.

## Creating Your Own Color Tables

First, I'll show you how to construct a simple color table. Then I'll show you how to extend the ideas behind the simple color table to construct any kind of color table you like.

Suppose we want a color table that runs from a yellow color in the first index to a red color in the last index. In terms of color triples, we want a color table that goes from [255, 255, 0] to [255, 0, 0]. You already know that a color table is made up of three vectors, containing the values for the red, green, and blue portion of a specific color. And, in most color tables, we would like a smooth progression from one value to the next, until we reach the final value.

What would constitute a smooth progression of colors? We see that for each of the red, green, and blue vectors we must go from the starting value in that vector to the ending value in that vector. And we must do it in some arbitrary number of steps that will be the size of our color vector. We can write a general expression for the vector that looks like this:

```
vector = beginNum + ((endNum - beginNum) * scaleFactor)
```

where we define the beginning number, the ending number, and the scale factor, which will depend upon the number of steps we want to take in getting from the beginning to the ending number.

Suppose we define these quantities like this:

```
IDL> beginNum = 10.0
IDL> endNum = 20.0
IDL> steps = 5
IDL> scaleFactor = FIndGen(steps) / (steps -1)
```

Then using the equation above, we print the vector values:

```
IDL> Print, beginNum + ((endNum = beginNum) * scaleFactor)
     10.0000   12.5000   15.0000   17.5000   20.0000
```

This looks right, so let's apply it to our color table problem. The red vector must go from 255 (the red value in the yellow color) to 255 (the red value in the red color). The green vector must go from 255 to 0. And the blue value must go from 0 to 0.

The red and blue vectors are extremely simple, since their values don't change. We can use the *Replicate* command to create those vectors. We will have to use our formula for the green vector, however. Here is the code to create a color table 200 elements in length.

```
IDL> steps = 200
IDL> rVec = Replicate(255, steps)
IDL> bVec = Replicate(0, steps)
IDL> scaleFactor = FIndgen(steps) / (steps - 1)
IDL> beginNum = 255 & endNum = 0
IDL> gVec = beginNum + ((endNum - beginNum) * scaleFactor)
```

Finally, load the color table vectors you created with *TVLCT*, and display an image that uses those 200 colors.

```
IDL> TVLCT, rVec, gVec, bVec
IDL> Window, XSize=200, YSize=40, Title='Color Table'
IDL> TVImage, BIndGen(steps) # Replicate(1B,40)
```


***Figure 10: A simple yellow to red color table.***

Using these principles you can construct as complicated a color table as you like. For example, suppose you want a 200 element color table that goes from yellow to red, as before, but you want it to go through a series of blue colors in the middle of the table. You simply break this down into two problems, each with 100 steps, that are similar to the first example. In other words, in 100 steps go from yellow [255, 255, 0] to blue [0, 0, 255], and then in 100 more steps from blue to red [255, 0, 0]. The code looks like this.

```
IDL> steps = 100
IDL> scaleFactor = FIndgen(steps) / (steps - 1)
```

Set up the first 100 steps, going from yellow to blue.

```
IDL> rVec = 255 + (0 - 255) * scaleFactor
IDL> gVec = 255 + (0 - 255) * scaleFactor
IDL> bVec = 0 + (255 - 0) * scaleFactor
```

Now do the second 100 steps, going from blue to red.

```
IDL> rVec = [rVec, 0 + (255 - 0) * scaleFactor]
IDL> gVec = [gVec, Replicate(0, steps)]
IDL> bVec = [bVec, 255 + (0 - 255) * scaleFactor]
```

Load the color vectors into the color table, and display an image using the colors.

```
IDL> TVLCT, rVec, gVec, bVec
IDL> Window, XSize=200, YSize=40, Title='Color Table'
IDL> TVImage, BIndGen(steps*2) # Replicate(1B,40)
```

Note that the IDL command *XPalette* allows you to create color tables by doing exactly this kind of interpolation between color values interactively. But I think it always helps to know what it is doing.

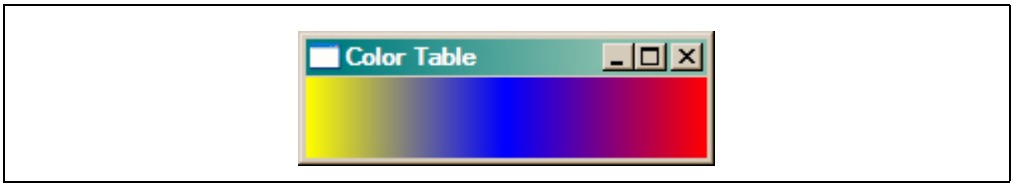


***Figure 11: A yellow to red color table, that passes through the color blue in the middle of the table.***

## Saving a Color Table

Before you can save a color table, you have to be able to obtain the RGB vectors that represent the color table. You may have created the vectors yourself, as above, or you may have created the color table by manipulating the color vectors interactively. (For example, you might have used *XColors*, *XLoadCT*, *XPalette*, or other tools to manipulate the color table vectors.)

If you manipulated the color table interactively, you can obtain the RGB vectors currently loaded in the color table by using the *Get* keyword to *TVLCT*. The vectors will be returned in the first three positional parameters. That is to say, the first three positional parameters will be output variables, rather than the input variables they are normally when you use the *TVLCT* command.

```
IDL> TVLCT, rVec, gVec, bVec, /Get
IDL> Help, rVec, gVec, bVec
     RVEC            BYTE      = Array[256]
     GVEC            BYTE      = Array[256]
     BVEC            BYTE      = Array[256]
```

These vectors contain as many elements as your color table (see *!D.Table_Size*), and will typically be 256 elements in length if you are using a 24-bit graphics card.

The simplest way to save these RGB vectors so they can be recalled later is to use the *Save* command. The vectors, including their names (*rVec*, *gVec*, and *bVec*), are saved in a machine-portable binary format (XDR) so they can be restored on any machine or platform running IDL.

```
IDL> Save, rVec, gVec, bVec, Filename='mycolortable.sav', $
       Description='Yellow-Blue-Red Color Table'
```

When you wish to use the color table, restore the variables and load them into the color table.

```
IDL> Restore, Filename='mycolortable.sav', Description=desc
IDL> IF desc NE '' THEN Print, desc
     Yellow-Blue-Red Color Table
IDL> TVLCT, rVec, gVec, bVec
```

Another way to save the vectors is to simply write them to a file. I recommend you use the XDR binary format and that you write the size of the vectors into the file first, so you can recreate the vectors in the correct size when you read them back out.

```
IDL> OpenW, lun, 'mycolortable.tbl', /Get_Lun, /XDR
IDL> WriteU, lun, N_Elements(rVec), rVec, gVec, bVec
IDL> Free_Lun, lun
```

To read the vectors out of the file, you write code similar to this.

```
IDL> OpenR, lun, 'mycolortable.tbl', /Get_Lun, /XDR
IDL> theSize = 0L
IDL> ReadU, lun, theSize
IDL> rVec = BytArr(theSize)
```

```
IDL> gVec = (bVec = rVec)
IDL> ReadU, lun, rVec, gVec, bVec
IDL> Free_Lun, lun
IDL> TVLCT, rVec, gVec, bVec
```

A third way to save a color table is to use the *ModifyCT* command to substitute your color table for one of the 41 color tables in the *colors1.tbl* file distributed with IDL. You will need administrator privileges to modify this file, but if you don't have them you can always copy this file to another file name and change the modified file. Load the modified file instead of the one distributed with IDL by using the *File* keyword with *LoadCT*, *XLoadCT*, *XColors*, etc.

Suppose, for some reason, we wished to have a 256 element color table in which the first 100 colors were gray-scale colors, and the next 156 colors were an orange color table, going from orange [255, 165, 0] to white [255, 255, 255]. We could construct such a color table like this:

```
IDL> LoadCT, 0, NColors=100 ; Indices 0 to 99
IDL> steps = 156
IDL> scaleFactor = FIndgen(steps) / (steps - 1)
IDL> rVec = Replicate(255, steps)
IDL> gVec = 165 + ((255 - 165) * scaleFactor)
IDL> bVec = 0 + ((255 - 165) * scaleFactor)
IDL> TVLCT, rVec, gVec, bVec, 100
```

And we could exchange this for the Prism color table (a vile, nasty color table, at least for teaching purposes) in the normal IDL distribution. The Prism color table is index number 6. (Have you made a backup copy of *color1.tbl* in case something goes drastically wrong in the next few minutes? I'd recommend it.) First, be sure you get the current color table vectors you just loaded into the color table.

```
IDL> TVLCT, r, g, b, /Get
IDL> ModifyCT, 6, 'HALF ORANGE', r, g, b
IDL> XColors
```

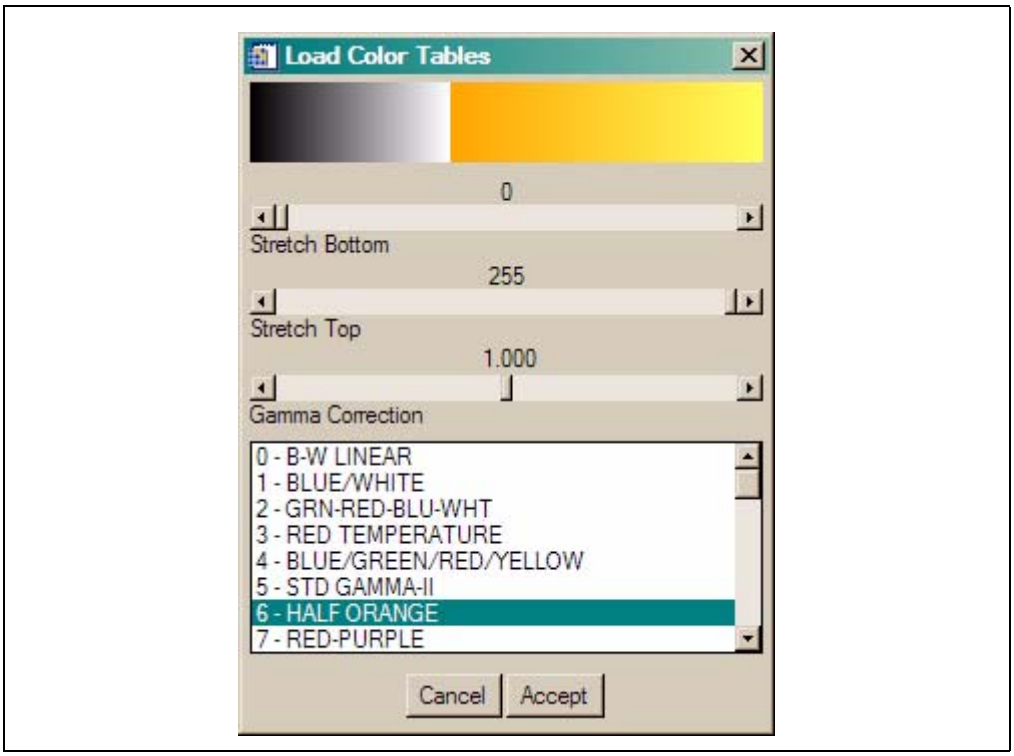


***Figure 12: XColors displaying the new color table in the modified colors1.tbl file.***

## Using Other Color Systems

While colors in IDL are always expressed as RGB values, and we must load RGB vectors into the color table, it is sometimes useful to express colors in other color systems. IDL also supports the HLS (hue, lightness, saturation) and HSV (hue, saturation, value) color systems. Colors in these systems are created with the *HLS* and *HSV* commands, respectively. Both of these commands load the color table that results from calling them into the current color table. And both return, in an optional parameter, the color system values converted to RGB values so these can be saved, reused, and so forth.

### HLS Color System

The HLS color system is sometimes also referred to as the HSL (hue, saturation, lightness or luminosity) or HSI (hue, saturation, intensity) color system. The system is typically draw as a double cone or spiral, and is (like the HSV system) a non-linear deformation of the normal RGB color cube. In IDL we specify the starting hue, which is a number from 0 to 360 (red equals 0, green equals 120, and blue equals 240) and indicate how many times we wish to loop through the color spiral. In addition, we specify the staring and ending lightness (a number from 0 to 100) and saturation (also a number from 0 to 100) values. The command looks like this:

```
HSL, light1, light2, sat1, sat2, hue, numloops, rgb
```

An image of the HSL color system cone or spiral is shown in Figure 13.[1] The *rgb* parameter is an output parameter that will contain a 256-by-3 array of RGB values that was loaded in the color table.

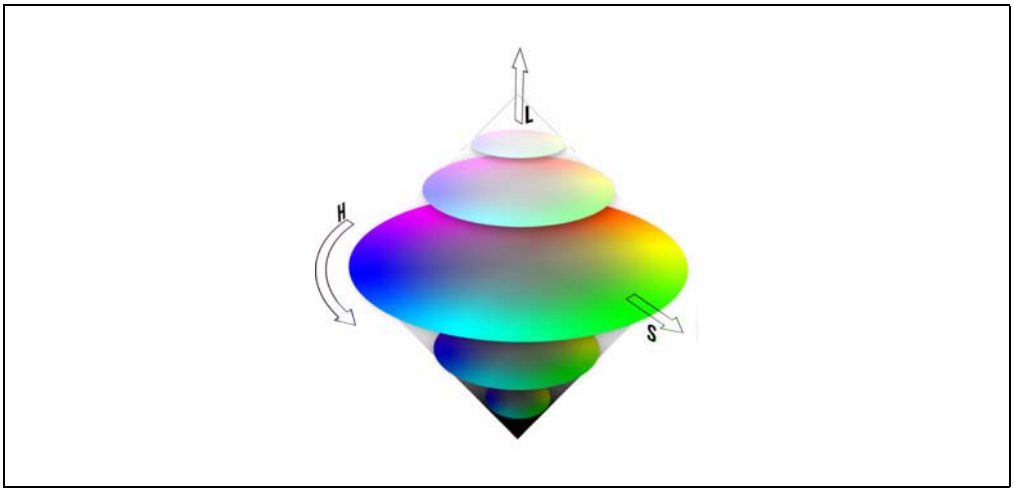


***Figure 13: A representation of the HSL color system cone or spiral.***

Here is code for a typical color table using the HSL color system. The output is shown in Figure 14.

```
IDL> HLS, 0, 100, 50, 100, 0, 1, rgb
```

### The HSV Color System

The HSV color system is often preferred by artists because of its similarities to the way humans perceive color. It is often visualized as a conical object in which the value is a number from the tip of the cone to the other end, saturation is the distance from the

1. Image downloaded from Wikipedia and used under the terms of the GNU Free Documentation License.

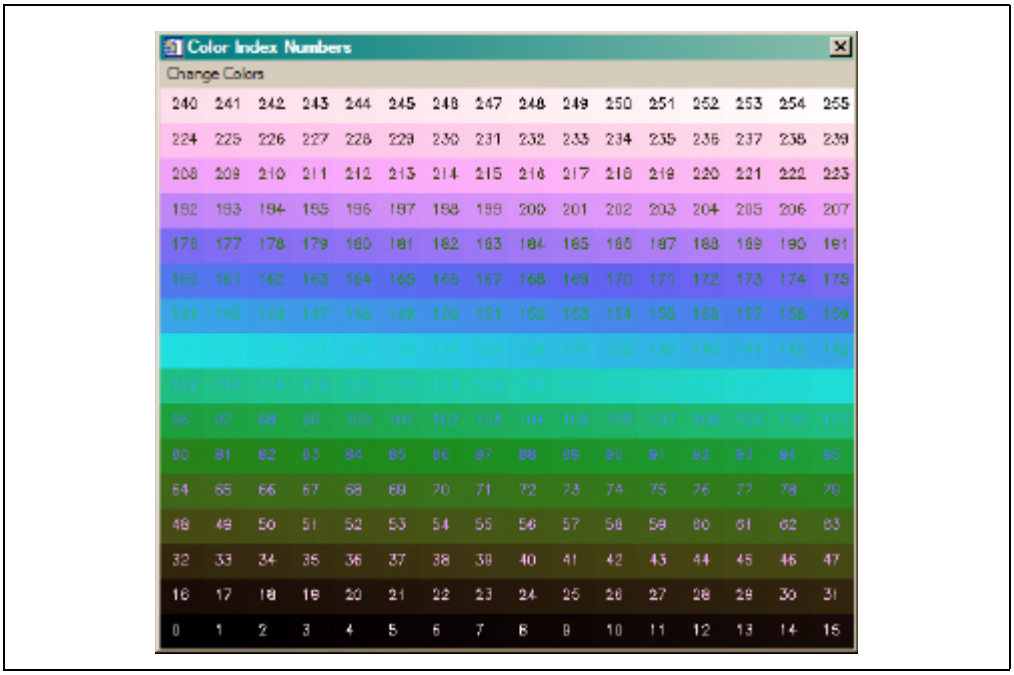


***Figure 14: A typical HLS color table created with the HLS command.***

axis of the cone, and hue is the rotation about the cone. You see a representation of the HSV color system in Figure 15.[1]

To produce a green temperature scale color table in the HSV color system, you would type a command like this:

```
IDL> HSV, 0, 100, 0, 100, 120, 0, rgb
```

You see the results of loading this color table in Figure 16.

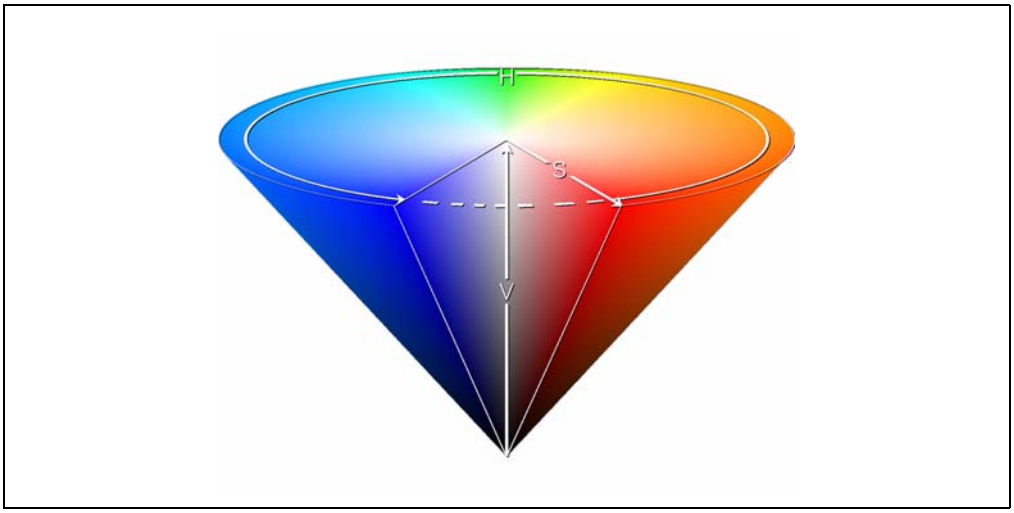


***Figure 15: A representation of the HSV color space.***

If you receive HSL or HSV values from elsewhere, and you wish to load them into an IDL color table, you can use the *Color_Convert* command to convert values in these systems into RGB colors, and visa versa. Your code will look something like this.

1. Image downloaded from Wikipedia and used under the terms of the GNU Free Documentation License.

```
Convert_Coord, hue, sat, light, r, g, b, /HSV_RGB
```

The first three parameters are input parameters, and the next three are output parameters containing the vectors after conversion. You must set the proper keyword to switch from one color system to another. See the on-line help for *Color_Convert* for more details.

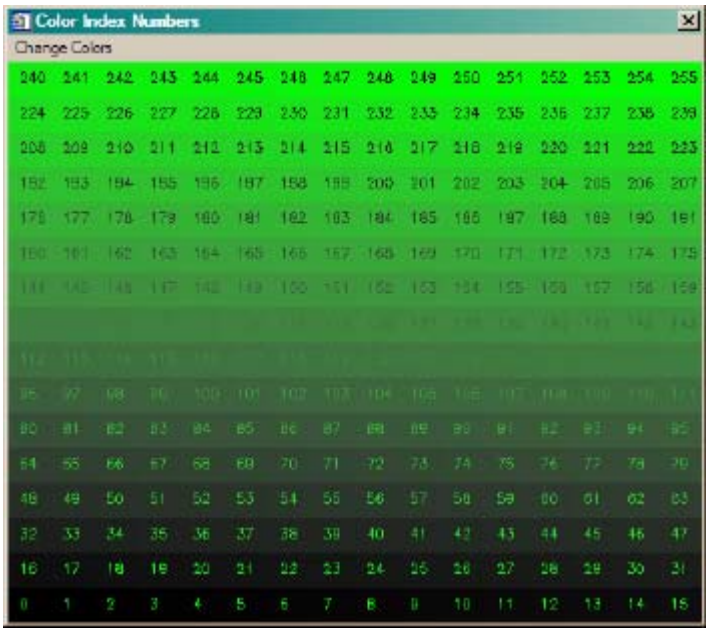


***Figure 16: A green temperature scale color table created with the HSV color system.***