Color History and Transition

The purpose of this chapter is to present a brief introduction to color in the traditional graphics system in IDL. (Color is handled differently and—some would say—more consistently in the object graphics system. But that is not a topic for this book.) According to participants in IDL programming classes, nothing is as consistently frustrating as trying to get color output to work correctly on the different graphics output devices supported by IDL (e.g., your Windows display, your colleague’s UNIX display, in the Z-graphics buffer, in PostScript output, etc.). This chapter will explain how color works in IDL and will introduce you to several color tools from the Coyote Library that will make it easier for you to write IDL programs that work in a device independent way.

Since about IDL 5.5 color handling has been in flux. Very few versions of IDL were consistent with earlier versions of the software. We were always learning something new. Programs that would run and display correctly on your computer would not display correctly on your colleague’s computer, and visa versa. Sometimes they wouldn’t display properly even if you were using the same computer operating system and version of IDL!

For a long time we were in a period of transition from 8-bit to 24-bit graphics cards. It was extremely important then to know how to write IDL programs that could coexist in both of these environments. But I haven’t run into an 8-bit graphics display in a very long time, so I pre-
sume nearly everyone has made the transition to 24-bit graphics cards successfully.

Unfortunately, we haven’t been able to escape the 8-bit environment or mind-set completely. We still have older graphics devices (e.g., older versions of the PostScript and Z-graphics devices) that are 8-bit devices. And, there is an enormous amount of legacy software written from an 8-bit point of view we have to deal with. (Even more alarming, there is a great deal of IDL software that continues to be written from an 8-bit point of view!) This is, to put it politely, a handicap when we want to use this software on 24-bit graphics displays.

So we still have problems to overcome. This chapter is designed to help you meet these problems head on, and to help you write more flexible traditional graphics programs in IDL. We will use what we learn here in examples throughout this book.

**UNIX Users Must Read This!**

If you are running IDL on a Windows computer, you can skip this section and ponder this question instead. How is it that an operating system so reviled by UNIX users can actually do some things right?

More on that (and on what Windows does wrong!) later, but this section is addressed to UNIX users in particular. If you were a UNIX user running versions of IDL prior to IDL 6.2, you were put, by default, into the X windows color environment, DirectColor, from which it was next to impossible to recover. Nothing you could do in that environment made much sense. Graphics windows were always flashing colors at you when you switched from one window to another. And the red on black color scheme in most IDL graphics windows, which made it almost impossible to see what was in the window, was thought to be 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 a memo!) was easy to miss. Even if they got the memo, getting their machines set up correctly was more like fooling around with 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 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. If you have an 8-bit depth or a `PseudoColor` visual name, then you are probably stuck with an ancient computer and I feel your pain, but you are in reasonable shape for colors. If your depth is 16, you are probably okay, but you will be confused when I talk about specific, vibrant colors. You will probably experience them as dull, lifeless things. If your depth is 24 and you have a `PseudoColor` visual name, and you are reading this book, it is probably because a few of your colleagues are complaining about your programs and would like you to know how to write programs that can coexist easily with 24-bit devices. 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 (if you have one) 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 by adding the following command to your IDL startup file. Be
sure to add it in front of any commands that might open an IDL graphics window.

    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 IDL traditional graphics commands work differently depending upon which model you chose to use in the IDL session. This problem is only compounded by the fact that some IDL programs (those written by many of your colleagues, no doubt, but some commonly used ones in IDL’s own library, too) will only work in one color model and not the other. And then, of course, colors work differently on Windows and UNIX machines, depending upon which version of IDL you are using. Sigh... It does take some time and experience to sort it all out.

The two color models are called decomposed color and indexed color. We sometimes 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 is to be used, and is set to 0 to indicate the indexed color model is to be used.

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

By default, when IDL starts up in a 24-bit environment (a TrueColor environment on UNIX, hopefully!), it will be using the decomposed color model. This is also true of Windows machines with a 24-bit graphics card. Or, another way to say this, color decomposition is on. But, what does this mean?
Colors Are Represented as Color Triples

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 vector a color triple. If we think of the values between 0 and 255 as color intensities, so that 0 represents none of the color, and 255 represents as much of the color as we can provide, then we have the possibility of specifying 256 times 256 times 256, or approximately 16.7 million, different 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 to be a representation of the visible world.

The two color models arise from how we select a color from this color palette. We might choose to select one of the 16.7 million colors directly, by specifying its color triple. We call this the decomposed color model (Figure 1). Or, we might choose to load a 256-element subset of these 16.7 million colors into a color table, and use the index number of the color table to select a specific color from the 256 choices we made available. If we do this, we specify a color triple by indicating which index in the color table is associated with the color we want to use. We call this the indexed color model (Figure 2).

If we decide to select our color directly, we must specify a color triple. But, rather than using a three-element vector, as the object graphics system does, in the traditional 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. You see an example of the decomposed color model in Figure 1.

Consider a green color, which can be represented by the color triple [53, 156, 83]. The first element is the red value, the second is the green value, and the third is the blue value. To construct a 24-bit value that can be decomposed into this color triple, we write code like this.

```
IDL> clr = [53, 156, 83]
IDL> greenColor = clr[0] + clr[1]*2L^8 + clr[2]*2L^16
IDL> Print, greenColor
5479477
```

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. The lowest 8 bits represent the red color, the middle 8 bits represent the green color, and the highest 8 bits represent the blue color.
If we wanted to express this green 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 100 with the TVLCT command, like this.

```IDL
IDL> TVLCT, 53, 156, 83, 100
IDL> greenColor = 100
```

The last argument to TVLCT is the color index where we are loading the color triple [53, 156, 83]. 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. You see this demonstrated in Figure 2.
We see now that the same green color can be represented as a 24-bit number (greenColor = 5479477L) or as a color index (greenColor = 100) 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 model (or, sometimes, the color decomposition state or color mode) 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 apro-
Working With Color in IDL

Most IDL users run into problems when they use color values that represent color index numbers in their code, but IDL is set to use the decomposed color model (which, remember, is the default color model), which interprets such 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 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, cgDemoData(1), Color=100
```

You see a red plot on a black background, as in Figure 3, where I have actually made the background color charcoal, so you can see the red plot a little better. Even so, this plot is extremely difficult to see. I have talked to users who believed this was the normal color scheme in IDL!

```
Figure 3: A mismatch between the color model in effect when a graphics command is issued and the way colors are specified can often end up with red plots on black backgrounds.
```

The solution, of course, is to match your color model with the color representation of your number (an indexed color model, for example, when you use the color index number 100 to represent a green color).

```
IDL> Device, Decomposed=0
```
Understanding IDL Color Models

IDL> Plot, Findgen(11), Color=100

Or, done the other way around.

IDL> Erase
IDL> Device, Decomposed=1
IDL> Plot, Findgen(11), Color=5479477L

Because so much IDL software has been written in the past with an 8-bit world view, many users find it advantageous to make sure they use the indexed color model when working with this software. This necessarily limits them to only 256 colors out of a palette of 16.7 million colors, but often this is enough. So you will see the following line in many IDL startup files.

Device, Decomposed=0 ; Start in indexed color mode.

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 wedding photo for a reference here.)

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 use the 16.7 million colors available to them 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 “green.”

Plot, Findgen(11), Color=100
Plot, Findgen(11), Color=5479477L

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 using 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='green'

And expect to find a green plot on my display no matter what decomposition state I am in or what 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 an “independent color” program, named cgColor. With cgColor I can write code like the following and I can always expect a green plot to appear on my display, no matter what color model is currently selected.

```
IDL> Plot, Findgen(11), Color=cgColor('green')
```

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

The program currently “knows” the names of about 200 colors. I chose these from a spectrum of colors to represent various drawing colors I would like to use in my own IDL programs, including the popular Brewer colors. But if you don’t like my colors, you can load your own colors from a text file that you can create and pass to cgColor. You can list the 200 colors in alphabetical order, like this.

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

Most of the colors you will want to use are probably in the list!

If you don’t know the name of a color to use, cgColor allows you to select a color interactively from a palette of colors. Use the SelectColor keyword like this.

```
IDL> Plot, Findgen(11), Color=cgColor(/SelectColor)
```

Or, you can simple use the PickColorName program, like this.

```
IDL> color = PickColorName()
```

You will see something that looks like the example in Figure 4.

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, the name is found in the names vector with the Where function. The location where the name is found is called the color index. The same index is used to find the corresponding red, green, and blue values in the color vectors to create the color triple.

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

Next, the program determines what color decomposition state is currently in effect for this IDL session. The `Get_Decomposed` keyword is used for this purpose.

```
Device, Get_Decomposed=currentState
```

If color decomposition is turned on, the program creates a 24-bit integer value from the color triple, and returns that as the result of the function. The Coyote Library program `Color24` is used to create the 24-bit value.

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

If, however, color decomposition is turned off, then the program loads the color at a particular color index number, and returns the color index number where the color was loaded to the user.

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

The 200 colors are designed to load themselves at unique indices at the top of the color table. This makes it possible to use various draw-
ing colors on your display and in PostScript files, for example, without having to think much about where those colors should be loaded in a color table. That is to say, this method of specifying drawing colors almost always 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 cgColor 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 cgColor like this.

```idl
color = cgColor('yellow', 240)
```

Note that the value of 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 (i.e., indexed color is turned on). Otherwise, colors are turned into 24-bit values that can be decomposed into the proper color values.

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

While the cgColor 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 drawing graphics to a PRINTER device, or sometimes when drawing a filled contour plot). The cgColor 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, /Silent
IDL> TVLCT, cgColor('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, cgColor(['teal', 'khaki', 'salmon'], $ /Triple), 201
```

You can see what colors you currently have loaded in your color table by using the Coyote Library program CIndex. You will see the yellow, teal, khaki, and salmon colors loaded at color indices 200 to 203.

```
IDL> CIndex
```

![Image of CIndex program](image)

*Figure 5: The CIndex program shows you the colors loaded in the current color table. Clicking anywhere inside the graphics window will refresh the window with the latest color table vectors.*

To see all the cgColor colors loaded in the color table, starting at color index 32, type the following command, then click your cursor inside the CIndex window to update its display.

```
IDL> TVLCT, cgColor(/All, /Triple), 32
```

The cgColor colors are loaded in indices 32 through 231.
Using cgColor With Color Table Indices

One of the major hurdles to overcome in making the transition to using decomposed color and having access to 16.7 million colors simultaneously is that so much software has been written assuming an indexed color model. This software loads a color table and then uses indices into that color table to specify the drawing colors.

For example, here is code that uses the three colors we just loaded into the color table.

```
IDL> TVLCT, cgColor(['teal', 'khaki', 'salmon'], $ /Triple), 201
IDL> Plot, cgDemoData(1), Background=202, Color=201
IDL> Oplot, cgDemoData(1), Color=203, PSym=2, $ Symsize=1.5
```

This kind of code only makes sense if we execute it using the indexed color model. If we execute it using the decomposed color model we get a window filled with a red color.

But, we can use cgColor to convert color table indices to any appropriate value by simply passing the color index as a string parameter. In other words, instead of passing the string “khaki” as the argument to cgColor, pass the string “202”. If cgColor reads the string as a “number” and needs a 24-bit value, it will create the value from the color triple loaded in the current color table at index 202.

In other words, these same commands written this way will work properly no matter which color model you are using.

```
IDL> TVLCT, cgColor(['teal', 'khaki', 'salmon'], $ /Triple), 201
IDL> Plot, cgDemoData(1), Background=cgColor('202'), $ Color=cgColor('201')
IDL> Oplot, cgDemoData(1), Color=cgColor('203'), $ PSym=2, Symsize=1.5
```

You will learn a great deal more about the benefits of using a program like cgColor 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.

Working Around a Color Bug on UNIX Machines

There is a very, well, interesting color bug that manifests itself on UNIX machines, including Macintosh computers, in IDL versions 7 and 8, and perhaps earlier. I mention it here because there is a possi-
bility you might run into it in the course of typing the commands in this book. You will notice you have a problem if you start an IDL session and run a graphics command that opens a graphics window with a white background color. The window will appear to be completely blank.

In fact, what is happening is that the system color variables, \texttt{!P.Background} and \texttt{!P.Color}, which determine the background and foreground drawing colors for plots, both get set to the color white. This happens on UNIX machines, when using the decomposed color model, and it only occurs for the first graphics window that opens in the IDL session.

You can solve the problem in one of two ways. First, simply closing the window and running the program that created the window again will solve the problem, because the problem only effects the first graphics window in the IDL session. Second, you can solve the problem in a more general way, by simply opening a graphics window and then deleting it in your IDL start-up file. The graphics window can be a pixmap window (a window that exists only in the video RAM of your computer, not on the display). I recommend all UNIX users insert the following two lines of IDL code into their start-up files.

\begin{verbatim}
Window, XSize=10, YSize=10, /Pixmap, /Free
WDelete, !D.Window
\end{verbatim}

This will keep you from ever seeing this particular bug.

\textbf{Color Models Also Affect Image Display}

Probably the number one reason so many IDL users limit 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 2D images with the IDL image display commands \texttt{TV} and \texttt{TVScl}. In particular, if you have the decomposed color model selected (it is the default color model, remember) and you load a color table, and display a 2D image, the image is \textit{not} displayed in color. This is enormously frustrating to users!

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

\begin{verbatim}
IDL> Device, Decomposed=1
IDL> LoadCT, 22
IDL> image = cgDemoData(2)
IDL> TV, image
\end{verbatim}
The image, which is supposed 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

What accounts for this? I’m not sure. I’ve always thought 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, in which all pixels have the same value in the red, green, and blue channel, 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 = cgDemoData(16)
IDL> Help, rose

Figure 6: Even with color tables loaded, 2D images are displayed in gray-scale unless the indexed color model is used.
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 as shown in Figure 8.

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

But if we use the indexed color model, the image is displayed correctly on Windows machines (using versions of IDL prior to IDL 7) only if the gray-scale color table is loaded. (Since IDL 7, a 24-bit image is displayed correctly in either color model.) It displays incorrectly if any other color table is loaded, as shown in Figure 9. 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. And, in any case, the same problem exists if you are displaying images in a PostScript file, as you will see later.

```
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 8: You have to use the color decomposition model to display 24-bit images correctly on Windows machines with color tables loaded, up until IDL 7. In IDL 7 and later, 24-bit images are always displayed correctly, no matter which color model is used.

Figure 9: On Windows machines, in IDL versions prior to IDL 7, 24-bit images were passed, incorrectly, through the color table vectors when in indexed color mode, resulting in incorrect color output. This is still true today for the PostScript device.

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 table vectors to look up different RGB values for the display of the image. Yikes!

For this reason, and many others (which you will learn about in more detail in “Creating Image Plots” on page 215), a great many IDL users no longer use the built-in TV or TVScl commands 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.

Another advantage of these commands is that they can automatically determine how the three color planes of a 24-bit image are interleaved, so you don’t have to worry about setting the True keyword to the correct interleave value with the TV command.

Two of the most popular of these substitute commands are cgImage, 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

I use cgImage almost exclusively in the code examples in this book to display images so I don’t have to worry about which color model or IDL version you are using, and so I can know we are both seeing the same colors when an image is displayed.

Setting the Color Model in a Device Independent Way

One of the goals of this book is to help you write device independent graphics programs. Another goal is to encourage you to use decomposed color whenever possible (i.e., pretty much always). These two goals are mutually exclusive for IDL programmers using versions of IDL older than IDL 7.1.1. (At least, if you define “device independent” as meaning the program produces identical results on the display, in a PostScript file, and in the Z-graphics buffer.)

The reason, of course, is that some devices are not compatible with the decomposed color model. The Z-graphics device didn’t become compatible until IDL 6.4. The PostScript device didn’t become compatible until IDL 7.1, and compatibility wasn’t fully implemented until IDL 7.1.1 when the Get Decomposed keyword worked correctly. The Printer
device, as well as older graphics devices (e.g., CGM, HP, etc.), are still not compatible with the decomposed color model.

This means that you can’t just issue the following command and expect it to work.

    IDL> Device, Decomposed=1

The *Decomposed* keyword presumes the device is capable of using either the color decomposed model or the indexed color model. It is not an allowed keyword for those devices that do not yet recognize decomposed color.

This makes it difficult to write programs that are going to run on computers other than your own, because you don’t always know what version of IDL the users of your programs have installed. In practice, it means writing a very long list of case statements to account for all the different devices and when they became compatible with 24-bit decomposed color.

As you can imagine, no one does this. Or, rather, if you are going to do it, you only want to do it one time and be done with it. I have done this for you in the *Coyote Library* command *SetDecomposedState*. This command allows you to set the color decomposition model of your choice (0 indicates indexed color, and 1 indicates decomposed color). The program takes into account all the device and version dependencies and sets the device to use decomposed color if it is possible to do so. For example, to use decomposed color, you type the following command. The *currentState* keyword returns the current color decomposition state before it is changed, so the color decomposition state can be returned to its previous state later.

    IDL> SetDecomposedState, 1, CurrentState=theState

After turning color decomposition on, we typically issue graphics commands, and, when we are finished, return to the previous state so we don’t change the user’s normal manner of working in IDL.

    IDL> SetDecomposedState, theState

Note that just because you tried to turn color decomposition on, doesn’t mean that it actually happened. We still have to write our programs to be color model independent, but at least we can take advantages of the color decomposed model if it is available. The primary advantages being, of course, that we have access to 16.7 million colors simultaneously and we don’t have to load drawing colors in the color table, contaminating the very colors we want to use for other...
pursposes. As time moves on, there will be fewer and fewer IDL users who cannot take advantage of decomposed color.

**Obtaining the Color Model State in a Device Independent Way**

There is also a problem obtaining the current color model state in a device independent way. That is to say, just as the device has to be 24-bit color compatible to be able to use the `Decomposed` keyword to set the color decomposition state, the device has to be 24-bit color compatible to use the `Get_Decomposed` keyword. But the situation is even more complicated than that.

For one thing, the PostScript device in IDL version 7.1 is 24-bit color compatible and accepts the `Decomposed` keyword, but the complimentary `Get_Decomposed` keyword didn't work until IDL 7.1.1. For another, the following command works differently on Windows machines than it does on UNIX machines.

```idl
IDL> Device, Get_Decomposed=currentState, Decomposed=1
```

On Windows machines, the current color decomposition state is returned, and then color decomposition is turned on for the graphics device. On UNIX machines, the color decomposition state is turned on for the graphics device, and then it is *this* state that is returned in the `currentState` variable. Yikes!

This means the only way to get the current state of the graphics device and then set the state to something else, in a device independent way, is to use two commands to do so.

```idl
IDL> Device, Get_Decomposed=currentState
IDL> Device, Decomposed=1
```

All of these dependencies have been encapsulated in the Coyote Library routine `GetDecomposedState`.

```idl
IDL> currentState = GetDecomposedState()
```

It is this routine that is called from within `SetDecomposedState` to obtain the current state of the graphics device and return it in the output keyword `CurrentState`.

---

**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 table index 22) into the 96 color indices, starting at color index 64. View the results by using the `CIndex` command. You see the result in Figure 10.

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

**Note:** 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 message off, use the `Silent` keyword to the `LoadCT` command.

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

**Figure 10:** Colors in IDL color tables can be restricted to portions of the color table by using the `NColors` and `Bottom` keywords to `LoadCT`. 
Loading Your Own Color Tables

There are times when the color tables supplied with IDL are not adequate for your purposes, or you wish to use other color tables. Both `LoadCT` and `XLoadCT` have a `File` keyword that you can use to load your own color table file.

For example, the Coyote Library that you downloaded to use with this book contains a file with 27 Brewer color tables. Brewer colors were designed by Cynthia Brewer (http://colorbrewer2.org/) to work particularly well on maps. This color table file is named `fsc_brewer.tbl`. Both diverging and sequential color tables are available. You can load the Brewer color tables with a command like this, which also uses the `Find_Resource_File` function from the Coyote Library to locate the file wherever it might be in your IDL path.

```idl
IDL> brewerFile = Find_Resource_File('fsc_brewer.tbl')
IDL> XLoadCT, File=brewerFile
IDL> LoadCT, 4, File=brewerFile
```

The IDL programs `XLoadCT` and `LoadCT` can work with just a single color table file at a time. Sometimes it is nice to be able to display either the IDL or Brewer (or some other!) color table file. The Coyote Library routines `XColors` and `cgLoadCT` are drop-in replacements for `XLoadCT` and `LoadCT` and have the Brewer color tables always available, in addition to the normal IDL color tables. We will sometimes use these programs in the examples in this book because of their additional functionality.

For example, if you wish to reverse a color table using `LoadCT`, you would do it like this.

```idl
IDL> LoadCT, 5
IDL> TVLCT, r, g, b, /Get
IDL> TVLCT, Reverse(r), Reverse(g), Reverse(b)
```

With `cgLoadCT`, this can be done in a single statement.

```idl
IDL> cgLoadCT, 5, /Reverse
```

With `XColors`, you can flip between IDL and Brewer color tables, reverse color tables on the fly, and use the program to communicate with other IDL programs in a variety of ways, all impossible to do with `XLoadCT`. You see in Figure 11 an example of how `XColors` appears on the display.
Creating Your Own Color Tables

It is easy in IDL to create 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

---

**Figure 11**: XColors has more features than XLoadCT and allows the user to access both the IDL and Brewer color tables simultaneously.
Working With Color in IDL

vector. We can write a general expression for the vector that looks like this.

\[
\text{vector} = \text{beginNum} + ((\text{endNum} - \text{beginNum}) \times \text{scaleFactor})
\]

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 = FINGen(steps) / (steps - 1)

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

idl> Print, beginNum + ((endNum - beginNum) \times \\
\text{scaleFactor})
\begin{align*}
10.0000 & \\
12.5000 & \\
15.0000 & \\
17.5000 & \\
20.0000 & \\
\end{align*}

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 256 elements in length.

idl> steps = 256
idl> rVec = Replicate(255, steps)
idl> bVec = Replicate(0, steps)
idl> scaleFactor = FINGen(steps) / (steps - 1)
idl> beginNum = 255 \& endNum = 0
idl> gVec = beginNum + ((endNum - beginNum) \times \\
\text{scaleFactor})

Finally, load the color table vectors you created with TVLCT, and display an image that uses these 256 colors.

idl> TVLCT, rVec, gVec, bVec
idl> Window, XSize=256, YSize=40, Title='Color Table'
idl> cgImage, BIndGen(steps) \# Replicate(1B,40)

Using these principles you can construct as complicated a color table as you like. For example, suppose you want a 256 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 128 steps, that are similar to the first example. In other words, in 128 steps go from yellow [255, 255, 0] to blue [0, 0, 255], and then in 128 more steps from blue to red [255, 0, 0]. The code looks like this.

```
IDL> steps = 128
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 128 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=256, YSize=40, Title='Color Table'
IDL> cgImage, BIndGen(steps*2) # Replicate(1B,40)
```
Note: 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.

It is even possible to do a piecewise interpolated color table. Consider for example, that you would like a color table that shows particular colors at particular color indices and you would like smooth transitions between them. Suppose you wanted the colors shown in Table 1. In this case, we use the IDL command Interpol to perform the piecewise interpolation from one value to the next. The code looks like this.

<table>
<thead>
<tr>
<th>Color Index</th>
<th>Red</th>
<th>Green</th>
<th>Blue</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>28</td>
<td>100</td>
<td>30</td>
<td>150</td>
</tr>
<tr>
<td>46</td>
<td>120</td>
<td>20</td>
<td>40</td>
</tr>
<tr>
<td>89</td>
<td>20</td>
<td>20</td>
<td>255</td>
</tr>
<tr>
<td>153</td>
<td>0</td>
<td>200</td>
<td>230</td>
</tr>
<tr>
<td>255</td>
<td>50</td>
<td>220</td>
<td>80</td>
</tr>
</tbody>
</table>

Table 1: A more complicated color table in which colors should flow smoothly from one specified index and color to the next.

IDL> r = Interpol([0, 100, 120, 20, 0, 50], [0, 28, 46, 89, 153, 255], Findgen(256))
IDL> g = Interpol([0, 30, 20, 20, 200, 220], [0, 28, 46, 89, 153, 255], Findgen(256))
IDL> b = Interpol([0, 150, 40, 255, 230, 50], [0, 28, 46, 89, 153, 255], Findgen(256))
IDL> TVLCT, r, g, b
IDL> Window, XSize=256, YSize=40, Title='Color Table'
IDL> cgImage, BIndGen(steps*2) # Replicate(1B,40)

You see the result in Figure 14.

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 by stretching the top or bottom of the color table, or by manipulating the gamma correction.)

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
IDL> TVLCT, rVec, gVec, bVec, /Get
IDL> Help, rVec, gVec, bVec
```

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
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
IDL> Restore, Filename='mycolortable.sav', $
Working With Color in IDL

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 that 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 either add a color table to the color table file (there is an upper limit of 256 color tables) or 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, cgLoadCT, etc.

Suppose, for some reason, we wished to have a 256 element color table in which the first 128 colors were gray-scale colors, and the next 128 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=128 ; Indices 0 to 127
IDL> steps = 128
IDL> scaleFactor = FIndgen(steps) / (steps - 1)
IDL> rVec = Replicate(255, steps)
IDL> gVec = 165 + ((255 - 165) * scaleFactor)
IDL> bVec = 0 + ((255 - 0) * scaleFactor)
IDL> TVLCT, rVec, gVec, bVec, 128

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 \textit{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, 'GREY-ORANGE', r, g, b
IDL> XColors

You see what the color table looks like in Figure 15.

\begin{figure}
\centering
\includegraphics[width=0.5\textwidth]{fig15.png}
\caption{The Prism color table is replaced by one of our own making using \textit{ModifyCT}.}
\end{figure}

If you prefer to add a color table to the file, rather than replacing a current color table, just give \textit{ModifyCT} the next highest number in the color table. In this case, number 41.

IDL> ModifyCT, 41, 'GREY-ORANGE', r, g, b
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 drawn as a double cone or spiral, and is (like the HSV system) a non-linear deformation of the normal RGB color cube. You see an example of an HSL color cone or spiral in Figure 16.

![Figure 16: A model of the hue, saturation, lightness color system.](image)

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 starting and ending lightness (a number from 0 to 100) and
saturation (also a number from 0 to 100) values. The command looks like this:

\[ \text{HSL, light1, light2, sat1, sat2, hue, numloops, rgb} \]

The `rgb` parameter is an output parameter that will contain a 256x3 array of RGB values that was loaded in the color table.

Here is code for a typical color table using the HSL color system. The output is shown in Figure 17.

\[ \text{IDL> HLS, 0, 100, 50, 100, 0, 1, rgb} \]

---

**Figure 17**: A color table built from specifying hue, saturation, and lightness.

---

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 flat base, saturation is the distance from the center axis of the cone, and hue is the rotation about the cone. You see a representation of the HSV color system in Figure 18.1
To produce a green temperature scale color table in the HSV color system, you would type a command like this.

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

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.

```idl
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.

---

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