

# 8-bit and 24-bit Color Graphics in PV-WAVE®

An In-Depth Look at How Color Graphics Work

**A White Paper by Visual Numerics, Inc.** December 2003

Visual Numerics, Inc. 12657 Alcosta Boulevard, Suite 450 San Ramon, CA 94583 USA

# 8-bit and 24-bit Color Graphics in PV-WAVE®

An In-Depth Look at How Color Graphics Work

by Visual Numerics, Inc.

Copyright © 2004 by Visual Numerics, Inc. All rights reserved. Printed in the United States of America.

**Publishing History:** 

December 2003

# Trademark Information

Visual Numerics and PV-WAVE are registered trademarks. IMSL, JMSL TS-WAVE, and JWAVE are trademarks of Visual Numerics, Inc., in the U.S. and other countries. All other product and company names are trademarks or registered trademarks of their respective owners.

The information contained in this document is subject to change without notice. Visual Numerics, Inc., makes no warranty of any kind with regard to this material, included, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Visual Numerics, Inc., shall not be liable for errors contained herein or for incidental, consequential, or other indirect damages in connection with the furnishing, performance, or use of this material.

# TABLE OF CONTENTS

Introduction			4
PART 1	Basic Color Concepts		
	1.1	Terminology	5
	1.2	More on Visual Classes	6
	1.3	More on Colortables	7
	1.4	8-bit Graphics Refresher	g
	1.5	24-bit Graphics Primer	g
PART 2	8-bi	t to 24-bit Code Changes	11
	2.1	Changing the Visual Class	11
	2.2	Referencing the Colortable	12
	2.3	The TV Procedure	13
	2.4	The TVRD Procedure	15
	2.5	More On Colortable O and WoColorConvert	16
	2.6	Summary	20
PART 3	Dire	ctColor vs. TrueColor*	21
	3.1	Changing the Colortable	21
	3.2	Using TVRD to Illustrate Their Differences	22
	3.3	Behind the Scenes	24
	3.4	DirectColor vs. TrueColor Schematic	26
	3.5	A Note About Colortables Under TrueColor	26
	3.6	TVRD's Numerical Accuracy	27
	3.7	Summary	29
Resources	and I	References	30

<sup>\*</sup> When the Windows operating system runs under a 24-bit display depth, the visual class is limited to TrueColor, so a direct comparison between these display types is not available to PV-WAVE users on Windows. However, there are numerous examples that help explain how TrueColor works and therefore it is still suggested reading for PV-WAVE users on Windows.

# Introduction

PV-WAVE users familiar with 8-bit graphics may initially find difficulty in taking advantage of PV-WAVE's capabilities on a 24-bit graphical display. Along with the increase in the number of colors available to the user comes an increase in the required level of understanding of how 24-bit and 8-bit graphics differ. The purpose of this document is to clarify those differences and provide understanding about how 24-bit graphics and PV-WAVE work together. There are numerous code examples, most of which can be simply copied-and-pasted from this document to the PV-WAVE prompt. It is suggested that you execute these code examples as the best way of "seeing for yourself." Code samples will be given in a Helvetica font. For example,

# LOADCT, 5 COLOR\_PALETTE

would be a code snippet to be copied-and-pasted to the PV-WAVE prompt. Where it is important to be able to reference a PV-WAVE command's output, the output will be preceded by a semicolon (;). For example,

#### PM, WoColorConvert(155)

# ; 10197915

Since the semicolon denotes a PV-WAVE comment, both lines can be copied-and-pasted to the PV-WAVE prompt without causing any syntax errors.

**NOTE:** It is assumed that you are familiar with the material covered in the PV-WAVE User's Guide, Chapter 11, *Using Color in Graphics Windows* and the PV-WAVE Reference Volume 3, Appendix B, *Output Devices and Windows Systems*, for the Win32 or X devices. Additionally, this document covers color issues in the context of traditional, non-widget PV-WAVE graphics. PV-WAVE Widgets have invariant colors on 8-bit and 24-bit displays. For example, widgets initialized with the following code

# topshell = WwInit('myWidget', 'test', workarea, Background='aquamarine')

have a background color of aquamarine, regardless of whether running on an 8-bit or 24-bit display.

# PART 1 Basic Color Concepts

#### 1.1 Terminology

It is helpful to establish common terminology used in discussing graphical output in the context of PV-WAVE. Chapter 7, *Color*, in Adrian Nye's <u>Xlib Programming Manual</u> [1] provides an excellent discussion on numerous color topics and establishes terminology commonly in use today. Although this terminology comes from the UNIX environment the ideas are the same in a Windows environment. Where confusion between the UNIX world and the Windows world is likely, the differences will be noted. Because VMS platforms use the same X Windows graphics server as UNIX platforms, references to UNIX will be synonymous with VMS.

**Colortable** – For PV-WAVE, a 256-element table. Each element in the table contains three values. These three values specify the intensity the red, green, and blue electron guns will display a given pixel. *Colortable* is synonymous with colormap.

**Depth** – The number of bits in memory used to specify a given pixel's value. For PV-WAVE, this is either 8 or 24.

DirectColor – A visual class with a depth of 24 in which each pixel on the screen is associated with a 24-bit value. This 24-bit value is decomposed into, or can be thought of as, three separate 8-bit values. One of these 8-bit values indexes into an array of red intensity values. Another indexes into an array of green intensity values, and the last 8-bit value indexes into an array of blue intensity values. These three intensities then define a given pixel's color. The red, green, and blue arrays that are being indexed into can be modified by an application and therefore the DirectColor visual class is often referred to as a dynamic visual class.

**PseudoColor** – A visual class with a depth of 8 in which each pixel on the screen is associated with an 8-bit value that references a location in a colortable. Each location in the colortable contains a triplet of values describing the intensity of the red, green and blue component of a given pixel. This colortable can be modified by an application and therefore the PseudoColor visual class is often referred to as a dynamic visual class.

**Raster graphics** – Graphics that are produced by bitmaps. Examples of PV-WAVE raster (or bitmap) graphics routines are **TV** and **IMAGE\_DISPLAY**. The input data to these routines are arrays of pixel values.

*TrueColor* - A visual class with a depth of 24 in which each pixel on the screen is associated with a 24-bit value. This 24-bit value is decomposed into, or can be thought of as, three separate 8-bit values. Unlike DirectColor, these three 8-bit values do not index into arrays of red, green, and blue intensity values, but rather, directly describe a pixel's red, green and blue intensities. Because the decomposed 24-bit value directly describes a pixel's color, there are no red, green and blue intensity arrays for an application to modify. For this reason, the TrueColor visual class is often referred to as a static visual class.

**Vector graphics** – Graphics that are produced by geometrical formulas or relationships between data. PV-WAVE vector graphics routines connect data points with lines or polygons. Examples of PV-WAVE vector graphics routines are **PLOT** and **vtkSURFACE**.

Visual Class - The type of colortable being used.

- \*\* Windows If the Windows operating system is using 24-bit graphics, the visual class that is being used approximates the TrueColor visual class.
- \*\* 32-bit Color Windows applications especially may refer to 32-bit color. In this case, there are still three 8-bit values describing the red, green and blue intensities found in each pixel. The "extra" 8-bits may be used to describe the level of transparency of the color (use of transparency in PV-WAVE is available through VTK). A more common reason for referring to 32-bit color is that the Accelerated Graphics Port on common PC hardware connects the graphics card to the CPU with a 32-bit wide channel. It can be more efficient to send pixel information as a 32-bit chunk, even though there may only be 24 bits of real information. For this paper's purposes, 32-bit and 24-bit color is synonymous.

#### 1.2 More On Visual Classes

The term *Visual Class* is important enough in PV-WAVE that is deserves discussion beyond the glossary definition. *Visual Class* is the name given to the type of display you are using. For example, DirectColor (24-bit depth), TrueColor (24-bit depth), and PseudoColor (8-bit depth) are the most commonly used visual classes, and are the types supported by PV-WAVE. To see what visual classes are available on your UNIX system, use the UNIX *xdpyinfo* command. An example might show the following:

```
screen #0:
```

dimensions: 1280x1024 pixels (320x240 millimeters)

resolution: 102x108 dots per inch

depths (4): 1, 8, 12, 24 root window id: 0x2b

depth of root window: 24 planes

default colormap: 0x20

default number of colormap cells: 256 preallocated pixels: black 0, white 16777215

number of visuals: 2 default visual id: 0x22

visual:

visual id: 0x21 class: PseudoColor depth: 8 planes

available colormap entries: 256
red, green, blue masks: 0x0, 0x0, 0x0
significant bits in color specification: 8 bits

visual:

visual id: 0x22 class: TrueColor depth: 24 planes available colormap entries: 256 per subfield red, green, blue masks: 0xff, 0xff00, 0xff0000 significant bits in color specification: 8 bits visual:

visual id: 0x23 class: DirectColor depth: 24 planes

available colormap entries: 256 per subfield red, green, blue masks: 0xff, 0xff00, 0xff0000 significant bits in color specification: 8 bits

We can see from the top line that this particular system has a screen resolution of 1280 pixels across by 1024 pixels high, and we have three visual classes available: PseudoColor, TrueColor, and DirectColor. On Windows 2000, supported visual classes can be identified through Start -> Control Panel -> Display and then click on the Settings tab. Under the Color drop down menu will be setting choices. Common choices might be 256 Colors, High Color (16 bit), and True Color (32-bit). The 256 Colors setting is essentially 8-bit PseudoColor and True Color (32 bit) is essentially 24-bit TrueColor. Recall that PV-WAVE supports visual depths of 8-bit (256 colors) and 24-bit.

PV-WAVE allows a user to use any visual class supported by their display, which in the Unix example means 24-bit TrueColor, 24-bit DirectColor, and 8-bit PseudoColor, and on a typical Windows system might mean PseudoColor and TrueColor. It is possible to switch between visual classes within a PV-WAVE session (only one visual class may be used at a time though) or have multiple PV-WAVE sessions using different visual classes. To switch between visual classes in PV-WAVE, use the DEVICE procedure. For example,

DEVICE, Pseudo=8
DEVICE, True=24

Note that the **DEVICE**, **/Close\_display** command will reset PV-WAVE to using the default visual class. The default for PV-WAVE is the DirectColor visual class. If DirectColor is not available, PV-WAVE will use the TrueColor visual class. In the event that the display does not support a visual class with a depth of 24, then PV-WAVE will use the 8-bit PseudoColor visual class, which is a standard choice available on all 8-bit displays. To determine which visual class a PV-WAVE session is currently using, issue the command **INFO**, **/Device**. This command must be issued after at least one graphics command has been issued, such as **TV**, **DIST(500)**. If such a graphics command has not been issued prior to the **INFO**, **/Device** call, PV-WAVE will not have had any contact with the operating system's window manager and therefore, what is reported may not be valid. More information about PV-WAVE devices and the DEVICE command can be found in Appendix B of the PV-WAVE Reference manual.

#### 1.3 More On Colortables

In section 1.1, a colortable was defined as a 256 element table, with each element containing a triplet of values defining the red, green and blue components of that table location's color. Let's examine this more closely using PV-WAVE. Enter the following commands at the PV-WAVE prompt:

LOADCT, 5
COLOR\_PALETTE

**COLOR\_PALETTE** displays what colors are associated with the currently loaded colortable. The "table" is actually three 256-element vectors, one each for the red, green, and blue colors comprising the colortable's elements. We can see what these vectors are by using the TVLCT command with the Get keyword.

#### TVLCT, red, green, blue, /Get

Looking at the color palette, we see that colortable location 114 is red and location 50 is blue. But colortable location 114 is really just the 114<sup>th</sup> value of the red vector, combined with the 114th value of the green vector, combined with the 114<sup>th</sup> value of the blue vector.

```
PRINT, red(114), green(114), blue(114); 255 5 0
```

So colortable location 114 has full red, a hint of green, and no blue, which makes a bright red. What about location 50, which the color palette shows is blue?

```
PRINT, red(50), green(50), blue(50); 14 0 250
```

Again, what we are really looking at is the 50<sup>th</sup> element in each of the red, green and blue vectors associated with colortable 5. In this case, we have a little red, no green, and lots of blue, which makes the blue color we see in the color palette at location 50.

Thus, for the rest of the discussion, when we say "colortable location 43", what we really are talking about is the composite of the  $43^{rd}$  value in each of the red, green and blue vectors associated with the given colortable. As a final example, look at the difference in the red, green, and blue vectors associated with colortable 13

table\_location = INDGEN(256)
LOADCT, 13
COLOR\_PALETTE
TVLCT, red, green, blue, /Get
PM, [[table\_location], [red], [green], [blue]]

compared to those vectors associated with colortable 4

LOADCT, 4
COLOR\_PALETTE
TVLCT, red, green, blue, /Get
PM, [[table\_location], [red], [green], [blue]]

#### 1.4 8-Bit Graphics Refresher

With 8-bit graphics, an application can display a maximum of 256 (2°) colors on the screen at once. This is because the video memory of the computer allocates 8 bits, or one byte, of memory for each pixel on the screen and is therefore only capable of representing each pixel with a value between 0 and 255. For every pixel on the screen, its associated 8-bit chunk of video memory contains a value referencing a location in a colortable. Therefore, a given pixel value can reference any one of 256 locations in a colortable. Each location in this colortable contains three numbers, again between 0 and 255. One number represents the intensity of the red component of the pixel's color, another number represents the intensity of the green component of the pixel, and the last number represents the intensity of the pixel. Since each location in the colortable may contain a unique triplet of red, green, and blue intensity values, each pixel may take on one of 256 possible colors. Although these three intensity values can combine to create approximately 16.7 million possible colors (256 possible red intensities x 256 possible green intensities x 256 possible blue intensities), the colortable may only contain 256 usable colors (since the video memory can only hold values between 0 and 255).

As an example, say colortable location 126 specifies the color white. Then colortable location 126 would contain the three values 255, 255, and 255 (full intensity red, green, and blue). As another example, the video driver finds in video memory a value of 43 associated with a particular pixel on the screen. It then looks at the colortable in location 43 and finds three numbers, say, 255, 0, and 255. The video driver uses these three values at colortable location 43 to tell the red, green, and blue electron guns in the monitor what intensities to display for that pixel location. In this example, any pixel with a value of 43 will be displayed as magenta (100% red + 0% green + 100% blue). For PV-WAVE, 8-bit graphics use the PseudoColor visual class.

#### 1.5 24-Bit Graphics Primer

With 24-bit graphics displays, video memory allocates 24 bits, or 3 bytes, for each pixel on the screen, and therefore each pixel can take on any one of a possible  $\sim 16.7$  million ( $2^{24}$ ) colors. Each 24-bit value is decomposed into 8 bits each for the red, green, and blue intensity values for that pixel. These triplets of 8-bit values are sometimes referred to as color planes, i.e., the red plane, green plane and blue plane. Thus, a 24-bit image can be thought of as three component images—a red, a green, and a blue image, which combine to create the "true color" picture. For example, to get a pixel to display as magenta (like the 8-bit example in section 1.4, where magenta can be formed by combining100% red + 0% green + 100% blue), we would specify a color value of 16711935, more clearly represented in hexadecimal notation as FF00FF. Note that in the hexadecimal notation, the rightmost two digits, (FF = 255) specify the red intensity, the two center digits (00 = 0) specify the green intensity, and the leftmost two digits (FF = 255) specify the blue intensity. So we can think of this 24-bit value as composed of three 8-bit parts. This is a subtle point that deserves further explanation.

On a 24-bit display, we might think that whenever we specify a plot to have the color FF00FF we will get a magenta colored plot. However, the color we will actually get depends on what colortable is loaded into PV-WAVE, because 24-bit color references are still passed through PV-WAVE's 256-element (8-bit) colortable. Recall from section 1.3 that when we specify a color with an 8-bit value (ranging 0-255), like 114, we are just combining the 114<sup>th</sup> element of each of the red, green, and blue arrays associated with the current colortable, producing the color we see from **COLOR\_PALETTE** at

location 114. Now, what happens if we give a 24-bit color reference, like FF00FF? This 24-bit value is decomposed into three 8-bit components, FF, 00, and FF, which in decimal is 255, 0, 255. These three 8-bit components are then used to reference the red, green, and blue arrays associated with the currently loaded colortable. In this example, we would get the color associated with combining the 255th element of the red array, with the Oth element of the green array, and the 255th element of the blue array. Notice that in the 8-bit case, the color value we specify references the same elements in each of the red, green, and blue color arrays (i.e., the color value of 97 references the 97th element in each of the arrays). In the 24-bit case, we can combine different elements of different arrays (i.e. the 20th element of the red array, plus the 97th element of the green array, plus the 231st value of the blue array), and this is what gives the 2<sup>24</sup> color choices. The important point though is that for both 8-bit and 24-bit displays, the values in these red, green and blue arrays depend on what 256-element PV-WAVE colortable is loaded. Thus, while intuitively a color reference of FF00FF means magenta (full red plus no green plus full blue), we must remember that PV-WAVE decomposes this 24-bit value into three 8-bit references into the red, green, and blue color arrays associated with a given colortable. Depending on what colortable is loaded, FF00FF may generate magenta or it may not. Sections 2.2 and 2.5 go into detailed examples of how 24-bit versus 8-bit color referencing works in PV-WAVE.

Although a 24-bit display takes up three times as much video memory as an 8-bit system, there is no restriction on the number of concurrent colors allowed on the screen except for the number of pixels on the screen itself. For example, a display with  $1280 \times 1024$  pixel resolution contains 1,310,720 pixels. If each pixel is assigned a unique color, 1,310,720 colors would be displayed at once, far less than the  $\sim 16.7$  million colors that are possible.

# PART 2 8-bit to 24-bit Code Changes

Part 2 concerns understanding code changes that are often necessary when porting PV-WAVE code from 8-bit displays to 24-bit displays. Before discussing these common code changes, it is important to identify your graphics needs. Depending on the results you would like to see, changes to your existing code may involve modifying one line of code, a few, or it could involve modifying many lines.

# 2.1 Changing the Visual Class

If you have existing code that works perfectly on your 8-bit display and you simply want that code to behave the same on a 24-bit display, then adding the line

# DEVICE, Pseudo=8

to the beginning of your code is all that is needed (assuming an 8-bit PseudoColor visual class is available on your system). When your code is executed after this command, PV-WAVE will use an 8-bit visual class, and thus your code will behave as if the underlying display was strictly 8-bit.

# 2.2 Referencing the Colortable

If it is the case that you not only want your old code to work as expected on an 8-bit display but you also want to take advantage of the greater flexibility of 24-bit color, then you will need to modify the way you reference PV-WAVE colortables. Such changes will also apply to the case where you are using a display that only supports 24-bit graphics (most displays support both 8-bit and 24-bit graphics).

The most common necessary code change involves the way colors are specified in calls to PV-WAVE vector graphics procedures like **PLOT**. For example,

DEVICE, /Close\_display
DEVICE, Pseudo=8
TEK\_COLOR
COLOR\_PALETTE
WINDOW, /Free
PLOT, INDGEN(10), Color=3

generates a plot with a green line. However,

DEVICE, /Close\_display DEVICE, True=24 TEK\_COLOR COLOR\_PALETTE WINDOW, /Free PLOT, INDGEN(10), Color=3 generates a seemingly blank screen. What has happened? In the 8-bit case, Color=3 tells PV-WAVE to use the color associated with location 3 in its colortable to color the plot line, which we see is green. In the 24-bit case, PV-WAVE uses 3 to index not into a 256-element colortable but into a set of values ranging  $[0-(2^{24}-1)]$ . Recall from section 1.2 a line of output from the Unix command *xdpyinfo* 

preallocated pixels: black 0, white 16777215

In this range of values, the value of 3 is nearly black and thus we cannot see the plot against the black background. This issue represents the most pervasive type of code change needed when converting from 8-bit to 24-bit graphics. The solution can be seen in the following change to the code.

# PLOT, INDGEN(10), Color=WoColorConvert(3)

This generates the expected green plot. What is WoColorConvert doing? To find out, let us first see what is happening in the 8-bit case. We can use the TVLCT command to get three vectors, red, green, and, blue, associated with the TEK\_COLOR colortable.

```
TVLCT, red, green, blue, /Get PM, red(3), green(3), blue(3); 0 255 0
```

As we might expect, the green color we see in colortable location 3 is defined by the triplet (0,255,0), meaning 0% red intensity, 100% green intensity, and 0% blue intensity. What is WoColorConvert(3)?

# PM, WoColorConvert(3)

; 197379

This large number actually corresponds to the same relative location in the spectrum of numbers ranging  $[0-(2^{24}-1)]$ , where  $2^{24}-1=16777215$ , as 3 exists in the spectrum of numbers ranging  $[0-(2^{8}-1)]$ , where  $2^{8}-1=255$ .

# PM, 197379.0/16777215.0

; 0.0117647

# PM, 3.0/255.0 ; 0.0117647

So, WoColorConvert takes an 8-bit value indexing into a colortable and scales that value into the range [0-(2<sup>24</sup>-1)]. Recalling section 1.5 above, PV-WAVE uses the 8-bit components of this 24-bit value to reference into the red, green, and blue arrays that define the currently loaded colortable (which by default is colortable 0 or Black/White linear). To make your code portable between 8-bit and 24-bit displays, use **WoColorConvert** in your color specifications. For example, under 8-bit color the following code produces plots that are identical.

DEVICE, /Close\_display
DEVICE, Pseudo=8
TEK\_COLOR
PLOT, INDGEN(10), Color=3
WAIT, 2
PLOT, INDGEN(10), Color=WoColorConvert(3)

So using WoColorConvert on an 8-bit system is not problematic. However, on a 24-bit system, we see that WoColorConvert is necessary.

DEVICE, True=24

PLOT, INDGEN(10), Color=3; plot is not visible

WAIT, 2

PLOT, INDGEN(10), Color=WoColorConvert(3) ; plot is the expected green color

Thus, the code

PLOT, INDGEN(10), Color=WoColorConvert(3)

will work as expected on either 8-bit or 24-bit systems. As a matter of convenience, we could do the following to reduce typing:

c = WoColorConvert(INDGEN(256)) ; scale the array values into the 24-bit range

TEK\_COLOR

PLOT, INDGEN(10), Color=c(3) ; rather than Color=WoColorConvert(3)

WAIT, 2

PLOT, INDGEN(10), Color=[c(2),c(3),c(4)]; rather than Color=WoColorConvert([2,3,4])

#### 2.3 The TV Procedure

The examples above used the vector graphics procedure **PLOT**, which has the keyword **Color** to reference the currently loaded colortable. What about raster graphics procedures like TV that do not have color keywords? Is there anything that needs to be done differently on an 8-bit vs. 24-bit display? The answer is "No" if the image array is 2D. The code below works the same on an 8-bit display as on a 24-bit display.

DEVICE, /Close
DEVICE, Pseudo=8
LOADCT, 14
img = DIST(500)
INFO, img
TVSCL, img
WAIT, 2
DEVICE, /Close
DEVICE, Direct=24
LOADCT, 14
TVSCL, img

However, when displaying a 3D image array on a 24-bit display, we need to use the True keyword with TV. We can use TV to display a 3D image array on an 8-bit display, but only the O<sup>th</sup> plane of the image (each plane being a 2D array) is considered. Let's make a 3D image array to see how TV displays it. First, let's define the three 2D image arrays that will comprise our 3D image array.

```
DEVICE, /Close
DEVICE, Pseudo=8
LOADCT, 11
zeroth_plane = BYTSCL( DIST(500) )
TV, zeroth_plane
WAIT, 2
first_plane = BYTSCL( HANNING(500,500) )
TV, first_plane
WAIT, 2
second_plane = REBIN( BYTSCL(LINDGEN(500)), 500, 500 )
TV, second_plane
WAIT, 2
```

Now that we have our three planes, concatenate the three 500x500 pixel arrays into one 500x500x3 array.

If we TV, img we should see a composite image of the three planes stacked upon each other. How is this 3D image array seen on an 8-bit display?

# TV, img

Now, comparing this with

# WINDOW, /Free & TVSCL, DIST(500)

we see the same image. Even though the variable img is a 3D image array, when we TV it on an 8-bit display, all we see is the  $0^{th}$  plane, or the DIST part of the image. Now, change to a 24-bit depth and try TVing img again.

DEVICE, /Close DEVICE, Direct=24 LOADCT, 11 TV, img

Again, we only see the DIST component of the 3D image array, even though we are on a 24-bit system. What we need to do is use the True keyword with TV to indicate that img is in fact a 3D array.

# TV, img, True=3

Now that we are on a 24-bit display and use the **True** keyword, we see the three images correctly superimposed. Why is the keyword **True** set to 3 and not some other number? Recall when we did the **INFO** on the img variable that its dimensions were 500x500x3. So this image array is just three 500x500 image arrays stacked upon each other. The **TV** procedure uses the **True** keyword to know where the image is interleaved, that is, how the three images in the 3D array are organized (for more information on *interleaving*, see the PV-WAVE Programmer's Guide). **True=3** specifies that the image is

MxNx3 (the "3" is in the third position in the array dimensions), whereas True=1 specifies that the image array is 3xMxN (the "3" is in the first position in the array dimensions). For example,

Notice the thin strip of color at the very bottom of the graphics window? After the **TRANSPOSE**, img is 500x3x500 so we need to set **True=2**.

ERASE
TV, img, True=2

#### 2.4 The TVRD Procedure

Analogous to the TV procedure discussed above, the **TVRD** procedure also has a **True** keyword. In the case of **TV**, we used the **True** keyword to specify that the image to be displayed was a 3D image array. For **TVRD**, we use the **True** keyword to specify that the image we want to capture should be represented by a 3D array. Here is an example:

So, the variable img is a 2D array of floating point values. We can use TVRD to capture a screen-shot of this image. The array that will contain this screen-shot can be either 2D or 3D.

```
img2D = TVRD() ; save the image as a 200x200 array
INFO, img2D
; IMG2D BYTE = Array(200, 200)
img3D = TVRD(True=1) ; save the image as a 3x200x200 array
INFO, img3D
; IMG3D BYTE = Array(3, 200, 200)
WINDOW, /Free, Xs=200, Ys=200, Title='2D TVRD image'
TV, img2d
WINDOW, /Free, Xs=200, Ys=200, Title='3D TVRD image'
TV, img3d, True=1
```

Note that we can use **TV** and not **TVSCL** on the image returned by **TVRD**. The original floating-point data variable, img, was produced by **DIST** and ranged between 0 and 140.007. We used **TVSCL** to display the data scaled into the range of 0-255 (the range of a byte) to make full use of PV-WAVE's 256-element colortable. When **TVRD** captures the image on the screen, it is capturing the byte value associated with each pixel we see, not the original data. Thus, **TVRD** returns a bitmap of the displayed data, and using **TV** to display this bitmap is sufficient. The **TVRD** procedure will be used in more examples in section 3.

#### 2.5 WoColorConvert and Colortable O

At this point in the discussion, having learned about the need to use **WoColorConvert** when specifying colors in vector graphic commands like **PLOT**, you may want to know more about why this function is necessary. This section further illustrates how **WoColorConvert** works, as well as the importance of colortable O.

Recalling the discussion in section 1.5, the 24-bit Graphics Primer, we may think of a 24-bit image as three images superimposed: a blue image laid over a green image, which is laid over a red image. In the same manner, we may think of a 24-bit value given to the Color keyword in **PLOT** as composed of three 8-bit numbers, one each corresponding to red, green, and blue intensity values. For example,

DEVICE, /Close\_display
DEVICE, Pseudo=8
LOADCT, 5
COLOR\_PALETTE
WINDOW, /Free
PLOT, INDGEN(10), Color=155

generates an orange plot. Now switch to a 24-bit visual class.

DEVICE, /Close\_display
DEVICE, True=24
LOADCT, 5
PLOT, INDGEN(10), Color=155

This generates a red plot, which is not what is expected given the current colortable that is loaded. Of course, **WoColorConvert** is needed...

PLOT, INDGEN(10), Color=WoColorConvert(155)
PM, WoColorConvert(155)
; 10197915

Note that if you find it easier to make sense of this 24-bit value using hexadecimal notation, you can use the PV-WAVE User library routine **LONG\_TO\_HEX**.

PM, LONG\_TO\_HEX(10197915); 9B9B9B

The following commands produce the same colored plot.

```
PLOT, INDGEN(10), Color=WoColorConvert(155), Title='WoColorConvert(155)' WAIT, 1
PLOT, INDGEN(10), Color='9b9b9b'x, Title='Hex value 9b9b9b' WAIT, 1
PLOT, INDGEN(10), Color=10197915, Title='Long integer value 10197915'
```

And it is orange, as expected. Let's find out what triplet is in colortable location 155 (i.e., what is the  $155^{\text{th}}$  value of the red, green, and blue vectors that define the colortable).

```
TVLCT, r,g,b, /Get
PM, r(155), g(155), b(155)
; 255 163 24
```

Red and green in an additive color system make yellow. Since there is more red than green, and very little blue, we get more of a reddish-yellow (i.e. orange) plot. What is the relationship between the value 10197915, the colortable triplet (255,163,24), and **WoColorConvert**? Again, we may think of the 24-bit value 10197915 as a composite of three 8-bit values. The first 8-bits specify a range of numbers [0-255] or [0 - (28-1)]. The next 8-bits cover a range [256 - 65535] or [28 - (216-1)]. Finally, the last 8-bits cover values in the range [65536 - 16777215] or [216 - (224-1)]. We can build any 24-bit value by combining these 8-bit values. So, when **WoColorConvert** converts 155 to a 24-bit value, here is what it is doing

```
PM, 155 + (155*2L^8) + (155*2L^16)
; 10197915
PM, WoColorConvert(155)
; 10197915
```

When we convert the triplet in colortable location 155, (255, 163, 24), into its 24-bit representation using the our formula

```
24bit-value = red + green*2^8 + blue*2^16 = red*256° + green*256° + blue*256°
```

we get

```
PM, 255 + (163*2L^8) + (24*2L^16); 1614847
```

However, a plot specified with that color is not orange (it is more of a light pink).

```
PLOT, INDGEN(10), Color=1614847
```

If we change the colortable to be colortable 0, and then plot with that same color value, we get the orange plot we are looking for

```
LOADCT, O
PLOT, INDGEN(10), Color=1614847
```

So, we can generate the same orange plot using

#### LOADCT, 5

PLOT, INDGEN(10), Color=10197915 ; WoColorConvert(155)=10197915

as we do using

#### LOADCT, O

PLOT, INDGEN(10), Color=1614847

Here is the relationship: First, let us look at the case when colortable 5 is loaded. Recall that the 24-bit value 10197915 in hex is 9B9B9B. 9B in decimal is  $155 (11*16^{\circ} + 9*16^{\circ})$ . So, this long value is decomposed into the triplet (155,155,155) and these values will be used to index into the red, green and blue vectors of the current colortable. Recall the code segment below.

LOADCT, 5
TVLCT, r,g,b, /Get

PM, r(155), g(155), b(155)

; 255 163 24

Thus, the 24-bit value10197915 is

- 1. Decomposed into the three 8-bit values 155, 155, 155.
- 2. These 8-bit values will be used to index into the red, green, and blue vectors associated with colortable 5
- **3.** Using the index triplet of (155,155,155), the red, green and blue values sent to the screen are 255, 163, and 24, respectively.

Now let us look at the case when colortable 0 is loaded. Colortable 0 is a PV-WAVE colortable that has all possible grey-scale colors in it from the 24-bit color spectrum, where grey is defined as any color with the same r,g,b values. Out of the  $\sim 16.7$  million colors that can be made by combining 256 different levels each of red, green, and blue, there are only 256 possible combinations that define a shade of grey.

(O , O , O ) - black

.

(128,128,128) - medium grey

•

(255,255,255) - white

Now compare

#### LOADCT, 5

TVLCT, r,g,b, /Get

 $index_num = INDGEN(256)$ 

PM, [[index\_num], [r], [g], [b]]

with

LOADCT, 0
TVLCT, r,g,b, /Get
PM, [[index\_num], [r], [g], [b]]

Notice that under colortable 0, the 3<sup>rd</sup> value of each vector is 3, and the 19<sup>th</sup> value of each vector is 19, etc., whereas there is no such correspondence between index and vector value under other colortables. Thus, when colortable 0 is used, a 24-bit color value *directly references a color*. Recalling the case with colortable 5, colortable location 155 (WoColorConvert(155)= 10197915) contained the triplet (255,163,24). So, under colortable 5 we can specify a color of 10197915 (9B9B9B in hex), which causes the rgb triplet (255,163,24) to be sent to the screen. Under colortable 0, what 24-bit value would be decomposed into (255,163,24) to give us the same orange color? Using our formula,

```
PM, 255 + (163*2L^8) + (24*2L^16) ; = r + g*256 + b*2562 ; 1614847 PM, LONG_TO_HEX(1614847) ; 18A3FF
```

In fact, there is exactly one r,g,b triplet that has a 24-bit representation of 1614847, and that is (255,163,24). No other r,g,b triplet is capable of representing 1614847. Under colortable 0, since the 255th element of the red vector is 255, and the 163rd element of the green vector is 163, and the 24th element of the blue vector is 24, when the 24-bit value 1614847 is decomposed into the triplet (255,163,24), 255, 163, 24 is actually what gets sent to the display. So, under colortable 0,

- 1. The 24-bit value is decomposed into three 8-bit values
- 2. These 8-bit values will be used to index into the red, green, and blue vectors associated with colortable 0
- 3. Since the values contained in the red, green, and blue vectors of colortable 0 correspond to their indices (i.e., the 3<sup>rd</sup> value is 3, the 135<sup>th</sup> value is 135 etc), the three 8-bit values are essentially sent straight to the screen

Under a non-zero color-table, this same triplet would get mapped to a new triplet, and *this* triplet is what would be sent to the display. For a final example, compare colortable 0 with 6

LOADCT, 0
TVLCT, r,g,b, /Get
PM, r(255), g(163), b(24)
; 255 163 24
PLOT, INDGEN(10), Color=1614847

Here, we see the orange plot we expect. Now try the same thing but after loading a PV-WAVE colortable other than  ${\bf 0}$ 

LOADCT, 6
TVLCT, r,g,b, /Get
PM, r(255), g(163), b(24)
; 0 114 0
PLOT, INDGEN(10), Color=1614847
WAIT, 3
DEVICE, /Close

Under colortable 6, when 1614847 is decomposed into the triplet (255,163,24), the triplet is mapped to the new triplet (0,114,0). As expected, we see a medium intensity green plot (no red, medium green, no blue).

# 2.6 Summary

To summarize the most important points from Part 2:

- On a 24-bit display, add the line **DEVICE**, **Pseudo=8** to the beginning of your code to make it run as if it was on an 8-bit display.
- Use WoColorConvert to reference colortables the same way between 8-bit and 24-bit visual classes.
- The TV procedure displays 8-bit images (2D image arrays) the same way on 8-bit and 24-bit displays. When displaying 24-bit images (3D image arrays) on a 24-bit display, the True keyword must be used to specify how the image is interleaved.
- The **TVRD** procedure also has a True keyword that can be used to save screenshots as 3D, 24-bit image arrays.
- To have a 24-bit value directly reference a color instead of referencing a location in PV-WAVE's colortable, have colortable 0 loaded (**LOADCT, 0**).

# PART 3 DirectColor vs. TrueColor

Now that we understand more about PV-WAVE colortables and 24-bit graphics, we might decide we want to write applications to take advantage of the vastly greater number of available colors that 24-bit graphics affords. The next question then is, under 24-bit depth, do we use the DirectColor or TrueColor visual class? To answer this question requires understanding the differences between these classes and this is best accomplished by example. Recall that the Windows operating system only has access to the TrueColor visual class under 24-bit depth. Although DirectColor and TrueColor can only be compared on a UNIX system, this section of the discussion is still relevant for users on Windows, as it explains the behavior of the TrueColor visual class.

# 3.1 Changing the Colortable

We'll start by looking at the biggest difference in behavior between DirectColor and TrueColor. This difference relates to how graphics are updated when there are changes to the colortable.

DEVICE, /Close\_display DEVICE, Direct\_color=24 LOADCT, 5 SHADE\_SURF, DIST(200)

We see the shaded surface image under colortable five's color scheme. To update the colors of the image we simply load a different colortable.

#### LOADCT, 4

Now try the same thing under TrueColor:

DEVICE, /Close\_display DEVICE, True\_color=24 LOADCT, 5 SHADE SURF, DIST(200)

However, changing the colortable does not modify the shaded surface

# LOADCT, 4

until the original graphics command is reissued.

# SHADE\_SURF, DIST(200)

This difference in behavior can be both convenient and limiting. For example, under DirectColor, we can update all graphics windows simultaneously to take on a new color scheme simply by loading a new colortable.

DEVICE, /Close\_display
DEVICE, Direct\_color=24
LOADCT, 6
SHADE\_SURF, HANNING(500,500) & WSHOW, !D.window
WINDOW, /Free & SHADE\_SURF, DIST(400) & WSHOW, !D.window
WAIT, 1
LOADCT, 5

However, we may want to display multiple graphics windows, each using a different colortable. This is not possible under DirectColor, but it is under TrueColor.

DEVICE, /Close\_display
DEVICE, True\_color=24
LOADCT, 6
SHADE\_SURF, HANNING(500,500) & WSHOW, !D.window
WINDOW, /Free
LOADCT, 5 & SHADE\_SURF, DIST(400) & WSHOW, !D.window

Of course, under TrueColor, if we do want the graphics window with the hanning surface to take on the colortable 5, we'll have to reissue the **SHADE SURF, HANNING(500,500)** command.

# 3.2 Using TVRD to Illustrate Their Differences

The example below will help explain why PV-WAVE behaves differently under TrueColor versus DirectColor. From your UNIX shell prompt, cd to the VNI\_DIR/image-1\_0/data directory (where VNI\_DIR is where PV-WAVE is installed, for example, /usr/local/VNI) and then start PV-WAVE. Cut-and-paste the following code blocks into the PV-WAVE prompt. The blocks have comments to provide some explanation, but more discussion will follow the code.

1)

; change to the TrueColor visual class

DEVICE, /Close

DEVICE, True\_color=24

- ; read a jpeg as an image array, which contains all of the 24-bit color information a=IMAGE\_READ('rockclimber24.jpg')
- ; grab the pixels from the image associative array, a b=a('pixels')
- ; load the green-pink colortable

LOADCT, 10

COLOR\_PALETTE

; TV the pixels, and use the True=3 keyword since the pixels are image-interleaved WINDOW, /FREE

TV, b, True=3

; The image is displayed with the green-pink color map applied to it

```
tell PV-WAVE to not pass the pixels through its colortable
DEVICE, Bypass_translation=1
WINDOW, /Free
TV, b, True=3
   image is now displayed with the 'real-life' coloring
    Note for Windows users: The Bypass_translation keyword does not exist for the WIN32
    device. To see the 'real-life' coloring of the image, load colortable O
3)
    tell PV-WAVE to go back to using the translation table and take a screen-shot of the
    image that has the 'real-life' coloring
DEVICE, Bypass_translation=0
c=TVRD(True=3)
WINDOW, /Free
TV, c, True=3
   when displaying the screen-shot of the 'real-life' image the colors are incorrect, but
    notice that they are not the same as in 1)
4)
    let's try the screen-shot again but bypass PV-WAVE's color translation table
DEVICE, /Bypass
WINDOW, /Free
TV, b, True=3
c=TVRD(True=3)
; tv the captured image into the same window
TV, c, True=3
; the real-life colors are redisplayed
In step 1) we changed to the TrueColor device. If in 1) we replace
DEVICE, /Close
DEVICE, True_color=24
with
```

DEVICE, /Close
DEVICE, Direct=24

and run through the four code blocks, there will be no difference in any of the images: all images will have the green-pink colortable applied. Thus, DEVICE, Bypass\_translation=1 has an effect under TrueColor but is seemingly ignored under DirectColor. To further understand this behavior requires a closer look at these visual classes.

#### 3.3 Behind The Scenes

To understand what is happening requires knowledge about how the DirectColor and TrueColor visual classes work behind the scenes with the graphics hard-ware. Using the Xlib Programmer's Manual's Glossary [1], let's revisit the definitions of DirectColor and TrueColor:

DirectColor is a class of colormap in which a pixel value is decomposed into three separate subfields for indexing. The first subfield indexes an array to produce red intensity values. The second subfield indexes a second array to produce blue intensity values. The third subfield indexes a third array to produce green intensity values. The RGB (red, green, and blue) values in the colormap entry can be changed dynamically.

TrueColor can be viewed as a degenerate case of DirectColor in which the subfields in the pixel value directly encode the corresponding RGB values. That is, the colormap has predefined read-only RGB values.

When a DirectColor visual class is initialized, its colormap is not defined. PV-WAVE subsequently defines that colormap, and PV-WAVE images pass through this map before being displayed to the screen. This map can change dynamically, such that if PV-WAVE changes the colormap definition (i.e. **LOADCT, 4**), then all graphics instantaneously reflect the change, because as the graphics are being refreshed (at say, 85 times a second), the image in video memory is passed through this new mapping and the new color scheme is reflected. When a TrueColor visual class is initialized, it is created with an immutable, pre-defined, colormap. Under TrueColor, since PV-WAVE cannot actually modify the color-map as it does under DirectColor, it creates a translation table that serves a similar purpose. To make use of a PV-WAVE defined colortable under TrueColor, PV-WAVE needs to create an appropriate translation table that modifies the pixel values *before* they are sent to video memory. By doing so, PV-WAVE makes it look like the data is passed through the PV-WAVE colortable even though the colortable is statically defined under TrueColor. At a very general level, here is the order of operations under DirectColor and TrueColor:

#### **DirectColor**

- A. PV-WAVE sends an image to video memory
- B. The image in video memory is decomposed into RGB components
- **c.** The RGB components are passed through the colortable, which can be changed dynamically
- **D.** The new RGB values are sent to the display.

So, the color mapping happens after the data is read from the video buffer. As the monitor refreshes the display (at 75 or 85 Hz etc) steps B), C), and D) are repeated. Changes to the PV-WAVE colortable are thus immediately reflected in existing PV-WAVE graphics.

# TrueColor

- A. PV-WAVE passes an image through a translation table, and then to video memory
- B. The image in video memory is decomposed into RGB components
- c. The RGB components are sent to the display.

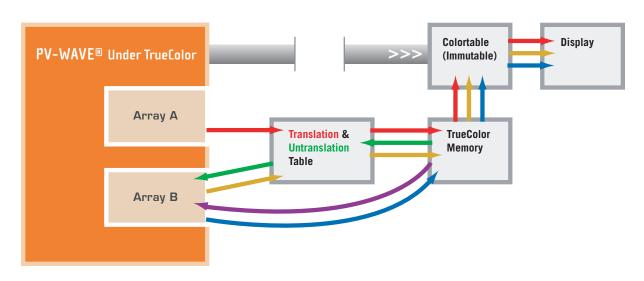
So, the color mapping happens before the data is sent to the video memory. When the graphics are being refreshed, steps B), and C) are repeated. Changes to the PV-WAVE colortable are not reflected in existing PV-WAVE graphics until the original PV-WAVE graphic commands are reissued in A), where the data will get passed through the new translation table associated with the new color scheme.

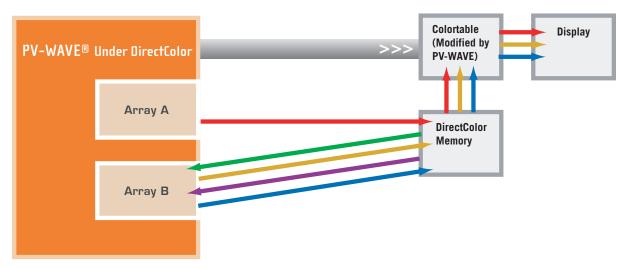
Let's look at the four examples in section 3.2 again. In 1), PV-WAVE takes the pixel data and passes it through a translation table and then to video memory. This data has been modified such that when displayed, it reflects the PV-WAVE colortable scheme as if it was passed through the colortable defined by PV-WAVE. In 2), we tell PV-WAVE to skip the translation and just send the data to video memory directly. Since we are working with a 3D image array from a jpeg, the pixel information already contains the "true" colors, and it can be read straight from video memory to the screen and look as expected. Now, in 3), we tell PV-WAVE to go back to using the translation before passing data to video memory. When we do the TVRD of the image, we are capturing a screen shot and are thus storing the actual RGB values that had originally been given to the display. If we try and TV this data, it is needlessly translated and then sent to video memory, so the redisplay is incorrect. What we should have done is done in step 4). Since the TVRD captured the actual RGB values being displayed, we have everything we need to redisplay the data correctly. So, we tell PV-WAVE to bypass the translation and just send the screen-shot image directly back to the screen. This ability to bypass the translation table is only possible under TrueColor. Under DirectColor, RGB values will always be passed through the current colormap before being sent to the display. Thus, the command

# **DEVICE**, /Bypass

can be used under either visual class, but under DirectColor it is ignored.

# 3.4 DirectColor vs. TrueColor Schematic





The colored arrows in the schematic correspond to the following code executed in this order:	To see this schematic in action, try the following code in PV-WAVE under DirectColor and TrueColor.		
PV-WAVE modifying the colortable  TV, a, True=3  b = TVRD(True=3)  TV, b, True=3  DEVICE, /Bypass_translation & b = TVRD(TRUE = 3)  DEVICE, /Bypass_translation & TV,b, True=3	CD, 'c:\vni\wave\demo\gallery3\data' img = IMAGE_READ ('rockclimber24.jpg') a = img('pixels') LOADCT, 5		

#### 3.5 A Note About Colortables Under TrueColor

We may look at the TrueColor schematic in 3.4 and wonder why the values from the TrueColor Memory box get passed through a colortable before being sent to the display, since the colortable under TrueColor does not modify the data. Recalling the definitions in section 1.1, the definition of TrueColor implied that there is no colortable; the values in video memory directly reference a given pixel's color. Thus, TrueColor is a "static" visual class because the colortable is immutable, and the reason the colortable cannot be modified is because the colortable does not really exist. However, another way we can think of the color-table under TrueColor was discussed in section 3.2, where the definition of TrueColor was cited from the Xlib Programmer's Manual's Glossary [1]. We may think of the colortable as a degenerate case in which what is passed in gets passed right back out.

In the TrueColor schematic in 3.4, the colortable is there to keep the schematic as symmetric as possible with DirectColor's schematic. This is to emphasize that the key difference between these visual classes for PV-WAVE is that TrueColor uses a PV-WAVE translation table to modify pixel values before they are sent to video memory, and DirectColor has no such translation table. Thus, whether to have the colortable included in the TrueColor schematic just depends on how we want to view the colortable.

#### 3.6 TVRD's Numerical Accuracy

In the examples of section 3.2, we were concerned with the *visual* accuracy of the **TVRD** function when used under DirectColor and TrueColor. That is, could a copy of the source graphic window be displayed faithful to the original image? We saw that under TrueColor, when a non-zero colortable was loaded we had to bypass the translation table to be able to display an accurate copy of the source image. We may wonder if, even though the copy *looks* like the source, is it really the same, pixel for pixel? As the final example of this discussion, and with the schematic from 3.4 in mind, we'll look at the interplay between TVRD, TV, and colortable 0, under DirectColor and TrueColor. To start the example, cd to the VNI\_DIR/image-1\_0/data directory. Run the following code block three times. Each time, five windows will appear showing different combinations of **TV, TVRD**, and use of the translation table. The example can be started in the following three ways, with the rest of the code pasted in afterward:

- 1) Under DirectColor with a non-zero colortable loaded DEVICE, /Close & DEVICE, Direct=24 & LOADCT, 5
- Under TrueColor with a non-zero colortable loaded DEVICE, /Close & DEVICE, True=24 & LOADCT, 5
- 3) Under TrueColor with colortable 0 loaded DEVICE, /Close & DEVICE, True=24 & LOADCT, 0

```
a = IMAGE_READ('rockclimber24.jpg')
b = a('pixels')
WINDOW, 0, Xs=476, Ys=316, Title='0 image under colortable 5'
WINDOW, 1, Xs=476, Ys=316, Title='1 tvrd image, no bypassing at all'
WINDOW, 2, Xs=476, Ys=316, Title='2 tvrd image, bypass on tv'
WINDOW, 3, Xs=476, Ys=316, Title='3 tvrd image, bypass on tvrd and tv'
WINDOW, 4, Xs=476, Ys=316, Title='4 tvrd image, bypass tvrd, no bypass on tv'
```

WSET, 0 TV, b, True=3 c=TVRD(True=3) WSET, 1 TV,c,True=3

DEVICE, /Bypass WSET, 2 TV,c,True=3

WSET, 0 d=TVRD(True=3) WSET, 3 TV,d,True=3

DEVICE, Bypass=0 WSET, 4 TV,d,True=3

PM, SAME(b,c), Title='Original array compared with TVRD array'
PM, SAME(b,d), Title='Original array compared with TVRD array, no trans'
PM, AVG(b), AVG(d), Title='Average pixel value of original array vs. bypassed TVRD array'

Now, let's review what we saw. In 1), which used DirectColor, all five windows had the same graphics, and the **SAME** function shows that the **TVRD**'d arrays are the same as the original array, regardless of whether the **Bypass\_translation** keyword is given to **DEVICE**. In fact, we know from the schematic in 3.4 that under DirectColor there is no translation table to bypass, and this keyword can be given without consequence. Under DirectColor the **TVRD**'d image is not only a faithful copy of the original image array visually, but also numerically.

In 2), which used TrueColor, there were a number of differences between the graphic windows that are best explained by referencing the schematic. Window 0 contains the original image under colortable 5. We **TVRD**'d this window and displayed the result in window 1, which does not look like what is in window 0. Since the translation table was not bypassed, we see from the schematic that when TVRD captures the image in window 0, it sends the result through the "untranslation" table where the result is stored in array c. This is the first problem: the video memory contains what we want to capture, but since the **Bypass** keyword was not given, the screen-captured image gets needlessly passed through the translation table before being stored in array c. Then, we TV the c array and it again gets passed through the translation table. In the schematic, it is the green and orange paths that are followed. In windows 2 and 4, only part of the TVRD/TV path was bypassed. In window 2, the green and blue paths were followed and in window 4, the purple and orange paths were followed. In either case, a translation of the original image is needlessly done and the resulting image is incorrect. Note that these windows have the same image, but they are incorrect in a different way than in window 1, which shows the result of the original image being needlessly translated not just once, but twice. Finally, window 3 shows the correct copy of window 0, and it is the purple and blue paths that are followed. In this case, the contents of the video memory are copied directly into array d, and this array is subsequently sent straight back to the video memory with no translation. However, even though window 3 shows an accurate visual copy of window 0, the respective image arrays (d and b) are slightly different numerically. This is because the original image array, b, had to go through the

translation table once before getting into video memory to be displayed in window 0. When we create the d array, we are getting the contents out of video memory and thus want to avoid any further translations.

In 3), under colortable 0 and TrueColor, PV-WAVE behaves the same as DirectColor in that the **TVRD'**d image arrays are the exact same, visually and numerically, as the original image array b. Because colortable 0 represents an invariant color mapping (i.e., what gets passed in gets passed right back out), the values in video memory displayed to the screen are exactly the same as the values in the array b. Regardless of whether the translation table was used or bypassed, arrays **c** and **d** are the same as **b**, visually and numerically.

# 3.7 Summary

To summarize the major differences between TrueColor and DirectColor:

- Under DirectColor we have the convenience of having all graphics windows automatically updated
  when new colortables are loaded, but this comes at the price of flexibility in that we can no longer
  simultaneously use different colortables in different graphics windows.
- Under TrueColor there is the opposite trade-off: the flexibility of having numerous simultaneously
  displayed colortables comes at the price of having to reissue all previous graphics commands for a
  colortable change to be systemic.
- The TVRD function, under DirectColor, always returns an exact copy of the source image array.
   Under TrueColor, the user must be careful to use colortable 0 and/or bypass the translation table when necessary.
- To see the 'real-life' colors of a 24-bit image, under either visual class, use colortable 0. Under TrueColor there is another option: you may have a non-zero colortable loaded and still see the 'real-life' colors of a 24-bit image by bypassing the translation table before displaying.

Because these visual classes have different trade-offs, rather than declaring which one is better, it is appropriate to consider the best contexts to use each of them. For example, in an application in which it is necessary to display multiple graphics windows simultaneously, each using a different colortable, the TrueColor visual class is the only choice that will work. Another application may require extensive use of TVRD, and this function is more straightforward to use under DirectColor. Yet another application may make extensive use of TVRD as well as having the need to display multiple colortables simultaneously. In this case, the DEVICE command could be used to switch back and forth between the TrueColor and DirectColor visual classes as necessary.

# Resources and References

Additional resources for learning about PV-WAVE and color can be found in:

- PV-WAVE Application Developer's Guide, Chapter 5: Using WAVE Widgets-Setting Colors and Fonts
- PV-WAVE Application Developer's Guide, Appendix D: Developing Portable Applications
- http://www.vni.com/tech/pvw/tn.html
- http://www.vni.com/tech/pvw/tips.html

# References

1. Nye, Adrian (1988), *Xlib Programming Manual for Version 11, Volume One,* O'Reilly and Associates, Inc., Newton, MA