OpenGL II: Windows Palettes in RGBA Mode

Dale Rogerson
Microsoft Developer Network Technology Group

December 4, 1994

Click to open or copy the files in the GLEasy sample application for this technical article.

Click to open or copy the files in the GLpal sample application for this technical article.

Abstract

If a program written for the Microsoft® Windows® operating system needs more than 16 colors and is running on an 8-bits-per-pixel (bpp) display adapter, the program must create and use a palette. OpenGL™ programs running on Windows NT™ or (eventually) Windows 95 are no exception. OpenGL imposes additional requirements on the colors and their locations on the palette in RGBA mode. The articles "OpenGL I: Quick Start" and "Windows NT OpenGL: Getting Started" in the MSDN Library cover the basics of using OpenGL in a Windows-based program and are required reading for this article. Two sample applications, GLEasy and GLpal, accompany this article.

Introduction

It would be nice to get together a small team of dedicated individuals, outfit them with the latest high-tech equipment, and airdrop them all over the world. These individuals would visit homes and offices everywhere, pull out users' existing unaccelerated 8-bits-per-pixel (bpp) graphics cards, and replace them with accelerated cards capable of at least 1024x768x2^24. This would be a community service paid by your tax dollars. All of this should happen because I hate palettes.

Palettes are confusing. They complicate software development. Yet for systems that can display only 8 bpp, palettes are a necessary evil. With 8 bpp, you can display only 256 colors. The chances that the 256 colors that you want are the same colors that trueSpace™, Corel® Draw, Mathcad, or some other application wants are pretty remote. In fact, a multiple-document interface (MDI) program that loads multiple 8-bpp device-independent bitmaps (DIBs) needs a different palette for each DIB it shows. Therefore, palettes are necessary until we all get 24-bpp display cards—an event that is at least a few years away. (16-bbp display cards don't have palettes either, but they have other limitations. In many cases, they can't display 200 levels of red, green, or blue, which is possible with an 8-bpp card and the correct palette.)

The moral of the story is: If a program needs more than the 16 system colors and must run on an 8-bpp display card, the program has to use a palette. Programs that use the OpenGL™ graphics library will require more than the 16 system colors to accurately shade objects for a realistic three-dimensional (3-D) display. Therefore, OpenGL programs that run on 8-bpp devices will need palettes.

This article examines the issues involved in palette management for OpenGL applications in RGBA mode. (For palette requirements in color index mode, see "OpenGL IV: Color Index Mode" in the MSDN Library.) In RGBA mode, colors are stored as red, green, blue, and alpha color components, and not as color indexes. The alpha color component controls color blending. An alpha value of 1.0 implies complete opacity, while an alpha value of 0.0 implies complete transparency.

In this article, my focus will be on the palette requirements introduced by the Windows NT™ implementation of OpenGL. This article does not provide a general discussion of palettes. For more information on palettes, please refer to the sources listed in the bibliography.

> **Note** In this article, the term "OpenGL" refers to the Windows NT implementation of OpenGL. Some limitations presented in this article are limitations of OpenGL, some are limitations of the generic pixel formats, and other limitations are associated with the Windows NT implementation of OpenGL.

PFD_NEED_PALETTE

If you remember "OpenGL I: Quick Start", I punted on the palette issue and told you to read this article. Here's a quick review of what I explained in that article.

OpenGL requires an active rendering context (RC). Without an active RC, OpenGL commands do nothing. Before you can create an RC and make it active, you must have a device context (DC) with a valid pixel format selected. Once again, I'll punt on pixel formats and refer you to Dennis Crain's article "Windows NT OpenGL: Getting Started" in the MSDN Library.

When you choose a pixel format, you specify the number of color bits per pixel desired. If the user's hardware supports 16, 24, or 32 bpps, a palette is not required. On the other hand, if the user's hardware supports only 8 bpps, you do need a palette. You can verify this by looking at the **dwFlags** field of the **PIXELFORMATDISCRIPTOR** structure after choosing or setting the palette. If **dwFlags** has the PFD_NEED_PALETTE flag set, you will need a palette.

You can use MYGL, the sample application that accompanies Dennis Crain's article, to enumerate through the available generic pixel formats and see that only the 8-bpp pixel formats have the PFD_NEED_PALETTE flag.

I'll illustrate this with some code from GLEasy, which I took from the OpenGL sample program GENGL (included in the Win32 ® SDK for Windows NT 3.5). In GLEasy, **CGLEasyView::CreateRGBPalette** creates the appropriate palette. Let's take a look at this function:

```
BOOL CGLEasyView::CreateRGBPalette(HDC hDC)
{
   //
   // Check to see if we need a palette.
   //
   PIXELFORMATDESCRIPTOR pfd;
   int n = GetPixelFormat(hDC);
   DescribePixelFormat(hDC, n, sizeof(PIXELFORMATDESCRIPTOR), &pfd);
   if (!(pfd.dwFlags & PFD_NEED_PALETTE)) return FALSE ;

   // Allocate a log pal and fill it with the color table info.
   LOGPALETTE* pPal = (LOGPALETTE*) malloc(sizeof(LOGPALETTE)
                    + 256 * sizeof(PALETTEENTRY));
   pPal->palVersion = 0x300; // Windows 3.0
   pPal->palNumEntries = 256; // table size

   //
   // Create palette.
   //
   ASSERT( pfd.cColorBits == 8) ;
   n = 1 << pfd.cColorBits;
   for (int i=0; i<n; i++)
   {
      pPal->palPalEntry[i].peRed =
             ComponentFromIndex(i, pfd.cRedBits, pfd.cRedShift);
      pPal->palPalEntry[i].peGreen =
             ComponentFromIndex(i, pfd.cGreenBits, pfd.cGreenShift);
      pPal->palPalEntry[i].peBlue =
             ComponentFromIndex(i, pfd.cBlueBits, pfd.cBlueShift);
      pPal->palPalEntry[i].peFlags = 0;
   }

   if ((pfd.cColorBits == 8)                           &&
       (pfd.cRedBits   == 3) && (pfd.cRedShift   == 0) &&
       (pfd.cGreenBits == 3) && (pfd.cGreenShift == 3) &&
       (pfd.cBlueBits  == 2) && (pfd.cBlueShift  == 6))
   {
      for (int j = 1 ; j <= 12 ; j++)
          pPal->palPalEntry[m_defaultOverride[j]] =
                                              m_defaultPalEntry[j];
   }

   if (m_pPal) delete m_pPal ;
   m_pPal = new CPalette ;

   BOOL bResult = m_pPal->CreatePalette(pPal);
   free (pPal);

   return bResult;
}
```

First, **CreateRGBPalette** determines whether a palette is needed by calling the new Win32 function **DescribePixelFormat** to get the current pixel format for the DC. **CreateRGBPalette** then checks the **dwFlags** field of the current pixel format to see whether PFD_NEED_PALETTE is set. If it isn't, the function returns without creating a palette.

Unfortunately, PFD_NEED_PALETTE is sometimes set, which means that we need to create a palette. In this case, we allocate a **LOGPALETTE** structure, **pPal**, using good old **malloc**. We fill in the version number and number of entries in the palette, then fill in the **palPalEntry** array with the colors. More on filling in the numbers in the next section.

After we fill in the **palPalEntry** array with 256 wonderful colors, we use **pPal** to create a palette. **CGLEasy::m_pPal**, which is a pointer to a **CPalette** object, points to this palette.

3-3-2 Palette

GLEasyView::CreateRGBPalette fills the 256 **PALETTEENTRY** structures. For OpenGL to render a picture correctly, **CreateRGBPalette** must fill these structures with the right colors. **CreateRGBPalette** uses the following fields in the **PIXELFORMATDESCRIPTOR** structure to fill in the palette correctly:

- **cRedBits**
- **cRedShift**
- **cGreenBits**
- **cGreenShift**
- **cBlueBits**
- **cBlueShift**

These fields determine how the 8 bits are divided into their red, green, and blue color components. **c*Bits** determines the number of bits allocated for a particular color. **c*Shift** specifies the location of that color within the 8 bits.

If you play with the MYGL sample, you will see that the generic formats supported by OpenGL, as implemented in Windows NT, have the following values for these fields:

Field	Value
cRedBits	3
cRedShift	0
cGreenBits	3
cGreenShift	3
cBlueBits	2
cBlueShift	6

As a result, the 8 bits are allocated as shown in Figure 1.

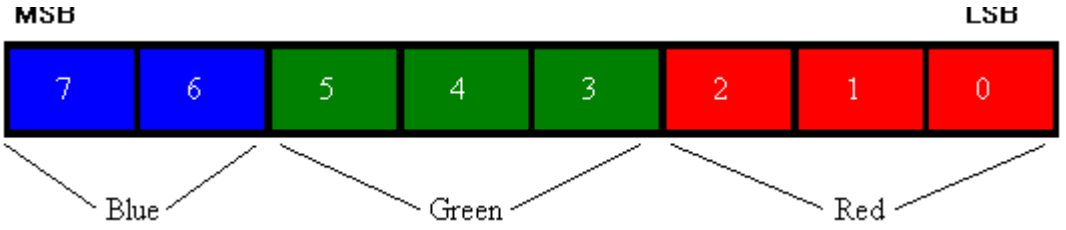

Figure 1. Allocation of color bits

For generic pixel formats that require palettes, the palette will be a 3-3-2 palette, which divides the 8 bits of color information into 3 bits for red, 3 bits for green, and 2 bits for blue. You may find it convenient to

think of a 3-3-2 palette as a color cube with red, green, and blue axes, as shown in Figure 2.

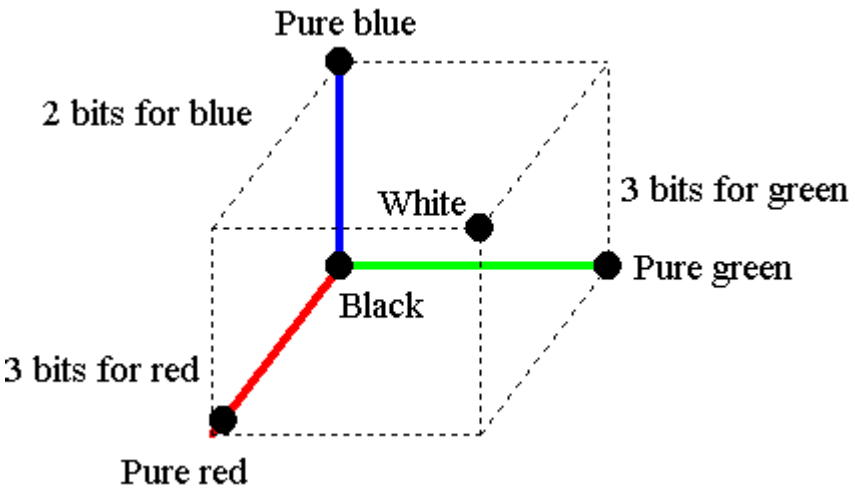

Figure 2. 3-3-2 color cube

In OpenGL, a color can be expressed using almost any numerical type, from an unsigned char (for example, **glColor3ub**) to a double (for example, **glColor3d**). Regardless of the units used, OpenGL converts the color to three floating-point numbers between 0.0 and 1.0 for internal use. After OpenGL performs all the calculations, it converts the color from its floating-point representation to a representation that can be displayed on the screen. If the current pixel format is one of the 8-bpp generic pixel formats, OpenGL converts the three floating-point numbers into one 8-bit value in the 3-3-2 format, and writes it to screen or bitmap memory.

On a palettized device, the value in display memory is the index into the palette. For the scene to be rendered correctly, the color stored in the palette at that index must be equivalent to the 3-3-2 color that its index represents. Let's look at a few examples.

Example 1.

The OpenGL command

```
glColor3d(1.0, 1.0, 1.0)
```

specifies the color white, which is the maximum amount of all colors. In a 3-3-2 scheme, the maximum amount of red is 111 binary, green is 111 binary, and blue is 11 binary. The color is written to memory as follows: the blue value, followed by the green value and the red value, resulting in 11111111 binary, or 255 decimal. The value written to memory is the index to the palette, so the 255th entry in the palette must store the color white, which would be (255, 255, 255).

Example 2.

The OpenGL command

```
glColor3d(1.0, 0.14, 0.6667)
```

specifies a nice purple color. To create this color, we need a maximum of red (111), 14 percent of the green (7 * .14 = 1, or 001 binary), and two-thirds of the available blue (3 * .6667 = 2, or 10 binary), so our 3-3-2 color would be 10001111 binary, or 143 decimal. The 143rd entry in our palette must have the same relative amounts of red, green, and blue. This would result in (255 * 1.0, 255 * 0.14, 255 * 0.6667), or (255, 36, 170). If this value is not in the 143rd palette entry, the wrong color will be displayed.

This is what the code in **CGLEasyView::CreateRGBPalette** does. It starts with 0 and iterates to 255, treating each number as an 8-bit, 3-3-2 value and converting the number to an RGB 8-bit triple to place in the palette. Most of this work is actually done by the **CGLEasyView::ComponentFromIndex** function and its supporting arrays **m_threeto8**, **m_twoto8**, and **m_oneto8**. As their names imply, these arrays

convert 3-bit, 2-bit, and 1-bit numbers to 8-bit numbers.

The **m_threeto8** array is interesting. It is specified as follows:

```
unsigned char CGLEasyView::m_threeto8[8] = {
   0, O111>>1, 0222>>1, 0333>>1, 0444>>1, 0555>>1, 0666>>1, 0377
};
```

In C and C++, a number with a leading zero is considered an octal number, so the above array actually stores the following numbers:

```
0, 36, 73, 109, 146, 182, 219, 255
```

These numbers are calculated by dividing 255 by the intervals covered by the bits we allocated (2^**c*bits** −1). For example, if **cRedBits** is 3, 255 / (2^3 −1)= 255/7 = 36.4. Repeatedly adding 36.4 will result in the array above. Notice that if you round 36.4 to an integer before adding it to the array, you get the following *different* array:

```
0, 36, 72, 108, 144, 180, 216, 252
```

You'll notice this rounding problem in the article "Advanced 3-D Graphics for Windows NT 3.5: The OpenGL Interface, Part II" by Jeff Prosise (*Microsoft Systems Journal* 9, November 1994; also in the MSDN Library Archive Edition). (This is a good article, and you should read it anyway.) Instead of using these cumbersome arrays, Jeff replaces the call to **ComponentFromIndex** with the following code :

```
byRedMask = (1 << pfd.cRedBits) - 1 ;
.
.
.
   lpPalette->palPalEntry[i].peRed =
      (((I >> pfd.cRedShift) & byRedMask) * 255) / byRedMask) ;
```

I like this code better than the arrays (personal bias); however, the integer division causes a rounding error, resulting in almost all of the colors having a component that is off by one. I can't tell the difference by looking at the palette, but it is incorrect. This problem can be fixed with the following change:

```
lpPalette->palPalEntry[i].peRed =
      (unsigned char)(((I >> pfd.cRedShift) & byRedMask)
      * 255) / (double)byRedMask + 0.5) ;
```

For the Sake of Simplicity. . .

In this discussion, I have simplified the process of how an OpenGL color appears on the screen. Most OpenGL scenes are rendered with some form of lighting or atmospheric effects that change color. In addition, OpenGL will, by default, dither colors to get a closer approximation of the desired effect. To stop OpenGL from dithering, use **glDisable(GL_DITHER)**.

I also did not mention that **CreateRGBPalette** can create palettes other than the 3-3-2 palette we have been discussing. The current implementation of OpenGL in Windows NT does not require you to create a different palette organization with the generic pixel formats. However, a hardware vendor can set PFD_NEED_PALETTE, **c*Bits**, and **c*Shift** to require a different organization. **CreateRGBPalette** is generic enough to handle these possibilities.

Yet another step I left out involves Windows® palettes. An application can build only a logical palette. The logical palette constructs a system palette, which the Windows operating system uses. Windows maps the colors in the logical palette to the system palette. In the system palette, the first and last ten colors are reserved for system use.

System Colors

If an application were allowed to use all of the colors in the palette, it could define all 256 colors as shades of red. As a result, all windows, text, menus, buttons, and everything else would be shades of red on red. You would click a red button on a red background, and enter red text with your red mouse cursor that you would be lucky even to see.

To keep at least the standard system window components looking normal, Windows reserves the first and last ten colors in the palette, as I mentioned earlier. These 20 colors include the 16 standard VGA colors. Most Windows-based applications use the standard 16 system colors instead of tinkering with palettes. This is why so many Windows-based applications look so boring.

An application can create a 256-color logical palette that does not include any of the 20 system colors; however, at most only 236 of these colors will end up in the system palette. Therefore, it doesn't make sense to request the additional 20 colors that you are not going to get.
CGLEasyView::CreateRGBPalette recognizes this situation. After filling the palette entries with the 3-3-2 palette, **CreateRGBPalette** puts the system colors in, as shown in the code below.

```
if ((pfd.cColorBits == 8) &&
   (pfd.cRedBits   == 3) && (pfd.cRedShift   == 0) &&
   (pfd.cGreenBits == 3) && (pfd.cGreenShift == 3) &&
   (pfd.cBlueBits  == 2) && (pfd.cBlueShift  == 6))
   {
      for (int j = 1 ; j <= 12 ; j++)
          pPal->palPalEntry[m_defaultOverride[j]] =
                                              m_defaultPalEntry[j];
   }
```

The GENGL and GLEasy sample applications use arrays, as shown below:

```
int CGLEasyView::m_defaultOverride[13] = {
    0, 3, 24, 27, 64, 67, 88, 173, 181, 236, 247, 164, 91
};

PALETTEENTRY CGLEasyView::m_defaultPalEntry[20] = {
    { 0,   0,   0,    0 }, //0
    { 0x80,0,   0,    0 },
    { 0,   0x80,0,    0 },
    { 0x80,0x80,0,    0 },
    { 0,   0,   0x80, 0 },
    { 0x80,0,   0x80, 0 },
    { 0,   0x80,0x80, 0 },
    { 0xC0,0xC0,0xC0, 0 }, //7

    { 192, 220, 192,  0 }, //8
    { 166, 202, 240,  0 },
    { 255, 251, 240,  0 },
    { 160, 160, 164,  0 }, //11

    { 0x80,0x80,0x80, 0 }, //12
    { 0xFF,0,   0,    0 },
    { 0,   0xFF,0,    0 },
    { 0xFF,0xFF,0,    0 },
    { 0,   0,   0xFF, 0 },
    { 0xFF,0,   0xFF, 0 },
    { 0,   0xFF,0xFF, 0 },
    { 0xFF,0xFF,0xFF, 0 }  //19
  };
```

The first array, **m_defaultOverride**, contains the indexes of the palette entries to be replaced with system colors. The second array, **m_defaultPalEntry**, contains the RGB values of the system colors to add to the palette. The **for** loop inserts only 12 of the 20 system colors into the logical palette. The remaining 8 colors are already in the 3-3-2 palette that was created, so there is no need to insert them.

The palette indexes stored in **m_defaultOverride** are determined by treating the colors as points in 3-D space and calculating the closest color in the 3-3-2 palette for each system color. The algorithm uses the standard distance formula (known as the *least-squares calculation*) that you learned in high school. The graphics device interface (GDI) **GetNearestPaletteIndex** function uses the same algorithm.

Don't use **GetNearestPaletteIndex** instead of the **m_defaultPalEntry** array to insert the system colors, because **GetNearestPaletteIndex** will replace index 255 instead of index 247 with the system color (255, 251, 240). It will also replace index 164 instead of index 156 with the color (128, 128, 128), although index 164 maps to the system color (160, 160, 164) much better.

If the 3-3-2 palette is not modified with the system colors, you will lose some of the colors. The system palette can accept only 236 non-system colors. In the 3-3-2 palette, only 8 of the 256 colors match the system colors, leaving 12 colors, which will be mapped either to the system colors or to one of the previous 236 logical colors. The problem is not that you lose 12 colors, but that you lose them at the end of the palette. A better approach is to try to map the system colors intelligently into the 3-3-2 palette, thus getting (we hope) 256 unique colors. (We are trying to have our cake and eat it, too.)

Again, I've simplified the discussion here. There are ways to gain control of all palette entries except the first entry (black) and the last entry (white), but this is not generally useful. See the sources in the bibliography for more information.

That covers the basics of building a palette for OpenGL. In the following sections, I would like to discuss a few other issues.

Identity Palettes

If you have read Nigel Thompson's book *Animation Techniques for Win32* (available from Microsoft Press®), you know that identity palettes are very important for fast blting.

An identity palette is a logical palette that is identical to the system palette. Not only does an identity palette have the same colors as the system palette, but the colors are in the same index order in the palette. To create an identity palette, Nigel first creates a palette. (He usually uses a 2-2-2 with a gray scale, but this doesn't really matter.) After selecting and realizing his palette, he gets the system palette entries and replaces his logical palette with the system palette. He now has an identity palette that is identical to the system palette. However, the color indexes in the identity palette do not match the indexes in the original logical palette, so he has to remap the colors in his DIBs. If he uses GDI calls, he uses the **PALETTERGB** macro to specify the colors.

OpenGL, as implemented in Windows NT, requires palettes in the 3-3-2 format. The identity palette is guaranteed to have system colors in the first and last 10 palette entries, which does not correspond to the 3-3-2 format. Therefore, it is not possible to have an identity palette with OpenGL when using RGBA color mode.

If you must have an identity palette, you might want to use color index mode instead of RGBA mode. In color index mode, you specify the palette index instead of the RGB values for the color you want. The downside of using color index mode is that some of OpenGL's effects, such as atmosphere, lighting, and blending, don't work very well. However, you can make the rendered scene look brighter and bolder by selecting the colors in the palette carefully. The standard 3-3-2 palette colors can look washed out or pastel. A 24-bpp display can fix all of these problems. (See my article, "OpenGL IV: Color Index Mode," in the MSDN Library for more information.)

Another solution is to render on a DIB section and let the DIB handle the color matching. More on this possibility in a later article.

Gamma Correction

When building the 3-3-2 palette, the palette index forms an 8-bit RGB color with the 3-3-2 format. The palette entry for a particular index contains a 24-bit RGB color with an 8-8-8 format. In the previous section, we did a linear conversion from 3 bits to 8 bits, and from 2 bits to 8 bits. Mathematically, this conversion results in the widest selection of colors. Biologically, however, it does not.

The human visual system (the eye, nervous system, and brain) is not equally sensitive to different color intensities. (This is one of the reasons we get good results with only 2 bits for blue.) The visual system does not interpret a linear increase in light intensity as a linear increase in perceived brightness. In fact, the human eye is more sensitive to the *ratios* of the intensities than it is to the absolute intensity values. Thus, the perceived difference between 0.1 and 0.11 is the same as the perceived difference between 0.5 and 0.55 (0.11/0.1 = 0.55/0.5 = 1.1). Therefore, using a logarithmic instead of a linear distribution results in better use of the available colors.

The process of correcting the intensity distribution is known as *gamma correction*. See the bibliography at the end of this article for more information on gamma correction.

However, as Gilman Wong of the Window NT OpenGL team pointed out to me, before we can consider perceived intensity, we need to ensure that the monitor will produce a linear intensity. As it turns out, most display-card/monitor combinations will not respond linearly to pixel values. The digital-to-analog converters (DACs) on the cards can also add non-linearities. This is the usual reason for gamma correction. Furthermore, trying to compensate for the human visual system is often impractical because it can vary from person to person. In fact, an array of physical and psychological factors (lighting conditions, fatigue, mood, and so on) can affect the perceived intensities even for a single person. If you use gamma

correction only to achieve linear monitor intensity, you're ahead of the game.

In the Win32 SDK for Windows NT 3.5 *OpenGL Programmer's Reference*, "Windows NT Extensions to Open GL," "About OpenGL on Windows NT," "OpenGL Color Modes and Windows Palette Management", the section "RGBA Mode and Palette Management" includes code for creating palettes for OpenGL. This code gamma-corrects the **m_threeto8** and **m_twoto8** arrays. The formula for gamma correction is:

$$f(v,g) = 255 \cdot \left(\frac{v}{255}\right)^{\frac{1.0}{g}}$$

where *v* is the value to be gamma-corrected, and *g* is the gamma-correction factor. A gamma of 1.0 provides an uncorrected linear distribution. The program uses a default gamma value of 1.4. The graph below shows the difference between the linear distribution and the gamma-corrected distribution using 1.4 for gamma. The horizontal axis is the index into the **m_threeto8** array.

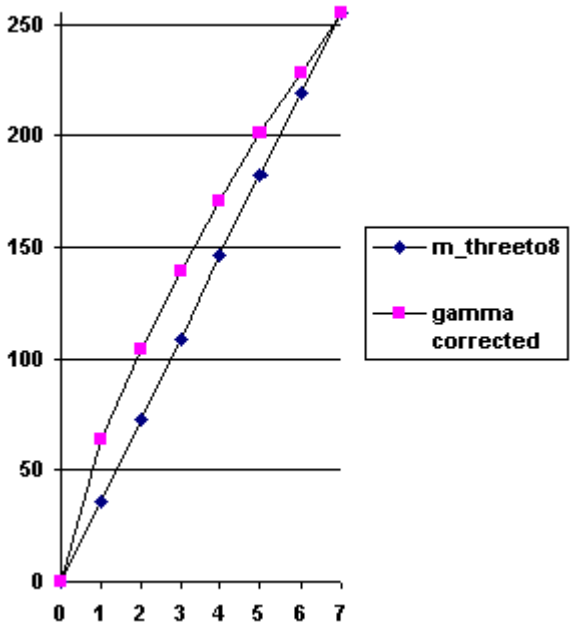

Figure 3. Graph of m_threeto8 array and gamma-corrected array

From the graph in Figure 3, it is evident that gamma correction biases the colors to higher intensities.

Should you use gamma-corrected colors? I tried it on a few scenes, and I really couldn't tell that much difference. I personally favor the uncorrected 3-3-2 format because the colors are more robust and less pastel. The next section discusses a way you can see palette differences, including differences in the gamma correction.

Examining the Palette

To understand palette interactions with OpenGL, I wrote an application called GLpal, which displays the logical palette using GDI and OpenGL. GLpal also shows the system palette.

The client area of the GLpal window is divided into four areas. Three of these areas display palettes, while the fourth area displays useful information.

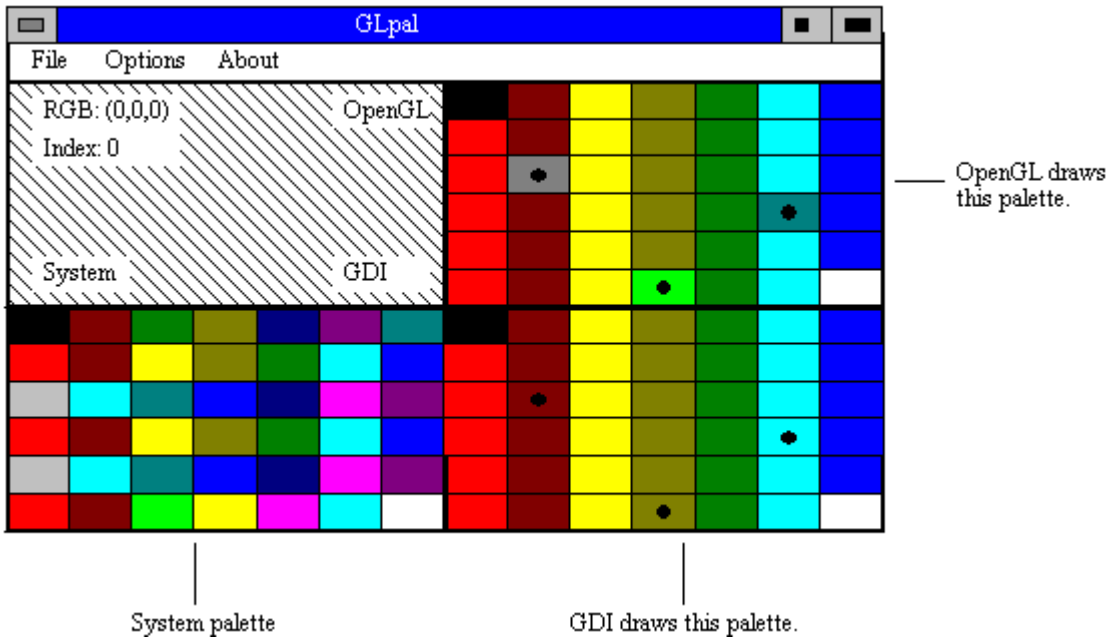

Figure 4. Simulated GLpal screen

Figure 4 simulates the GLpal screen. The real program displays 256 colors in each palette it displays. The system palette is displayed in the lower-left corner, the logical palette (drawn using OpenGL) is displayed in the upper-right corner, and the logical palette (drawn using GDI) is displayed in the lower-right corner.

To see the RGB value and the palette index for a color, press the left mouse button over the color, and drag the mouse to see other colors. These values are displayed in the upper-left corner of the window.

Drawing the Logical Palettes

The two logical palettes are drawn simultaneously by the **CGLpalView::DrawGlPalette** function. **GetPaletteEntries** fills a **PALETTEENTRY** array, **pal**, with the RGB values currently in the logical palette. These are the RGB values that we would like to display on the screen. GLpal displays squares filled with each color in the logical palette. Each square is drawn twice: first with GDI and then with OpenGL. To draw the GDI square, we use a brush that is created with the following command:

```
CBrush br ;
br.CreateSolidBrush(PALETTERGB(pal[iCurColor].peRed,
                               pal[iCurColor].peGreen,
                               pal[iCurColor].peBlue)) ;
```

CDC::Rectangle draws the square using this brush. Immediately after drawing the square with GDI, OpenGL draws the square (in the same color, we hope). The following OpenGL command sets the color using the RGB value from the **pal** array:

```
glColor3ub( pal[iCurColor].peRed,
            pal[iCurColor].peGreen,
            pal[iCurColor].peBlue );
```

The OpenGL **glColor3ub** function (Nigel calls it the "color thrub" command) accepts the color levels as unsigned bytes. (This is convenient because that is what the RGB values in the **PALETTEENTRY** structure are.) OpenGL internally converts these unsigned byte values to floating-point values.

DrawGlPalette should use the color **RGB(r,g,b)** when it executes **glColor3ub(r,g,b)**.

If you turn color tracing on (see the "Other GLpal Options" section), GLpal compares the RGB value for the GDI square with the RGB value for the OpenGL square. Colors that do not match are identified by a black dot in the middle of the square.

Drawing the System Palette

The **CGlpalView::DrawSysPalette** function draws the current system palette. GLpal creates a **LOGPALETTE** structure and fills the **peRed** field of each **PALETTEENTRY** structure with the index of that particular structure. The **peFlags** field is filled with PC_EXPLICIT. The **LOGPALETTE** structure creates a palette that contains the current system colors. Drawing the palette is left to GDI. In this case, we use the **PALETTEINDEX** macro instead of the **PALETTERGB** macro to specify the colors.

This is all kind of a joke. We simply wish to draw what is in the hardware palette, but this is not all that easy because GDI thinks only about the logical palette. The punch line to this joke is that we have to select the system palette in the background so that our application does not remap its own palette.

Changing the Palette

The Palette command in the GLpal Options menu allows you to change the palette created by GLpal. The Palette Options dialog box lets you choose between the 3-3-2 palette we have been discussing and the wash palette that Nigel Thompson uses in his book *Animation Techniques for Win32*. If you select the wash palette, you will see that OpenGL uses the wrong colors.

The other palette options you can change pertain only to the 3-3-2 palette. You can have the system colors inserted in the palette by checking the Add System Colors check box. Clear the check box and examine the end of the OpenGL palette. You will see several colors repeated. Click the left mouse button and drag to see the RGB values of the colors in the palettes.

You can also gamma-correct the palette using a gamma factor of 1.4, 1.8, or 2.0. A gamma of 1.0 is the same as not gamma-correcting the palette. As the gamma factor goes up, you will notice that the colors you request increasingly diverge from the colors you get, which is to be expected. The palette also becomes more pastel as the high intensities take over.

Other GLpal Options

Changing the palette is not the only option provided by GLpal; you can also change dithering, trace colors, and the selection of the palette before the **wglMakeCurrent** call.

Dither

The Dither option turns dithering on and off. To get more realistic shadings, OpenGL dithers the colors. Because dithering makes it difficult to determine the actual color of a square, I turn dithering off by default. Select the Dither command from the Options menu to enable dithering. In OpenGL, dithering is controlled by the following commands:

```
glEnable(GL_DITHER) ;
glDisable(GL_DITHER) ;
```

Trace Color

If you enable the Trace Color option in the Options menu, GLpal marks colors in the OpenGL palette that differ from the GDI palette. The colors are marked in both palettes with a small black dot. This feature is enabled by default. It doesn't make any sense to use the Trace Color option with dithering turned on because Trace Color compares only one pixel in each square. If dithering is on, the pixels within a square will be different colors.

Select Before wglMakeCurrent

You must select the palette in the DC before setting the current rendering context with **wglMakeCurrent**. To confirm this requirement, disable the Select Before wglMakeCurrent option in the Options menu. The palette will not be selected before the **wglMakeCurrent** call and will not be drawn correctly. I threw this option in to prove that you must select and realize your palette in the DC before setting the current rendering context.

Colors That Don't Match

Run GLpal. You will see that three colors in the OpenGL palette do not match the corresponding colors in the GDI palette. The colors are the following:

Index	GDI RGB Value	OpenGL RGB Value
164	(160,160,164)	(146,146,85)
236	(166,202,240)	(164,182,170)
247	(255,251,240)	(255,219,170)

These are the three colors that were changed when we added the system colors to the palette. The GDI RGB values are the values of the system colors we placed in the palette. If you clear the Add System Colors check box in the Palette Options dialog box, the system colors will not be added, and none of the colors in the two palettes will differ. However, some colors will be lost.

The colors differ because of the way the Windows NT implementation of OpenGL quantizes colors to 3-3-2 formatted, 8-bit values when rendering on a palettized device. Internally, OpenGL represents color intensities as floating-point numbers between 0.0 and 1.0. After OpenGL finishes all of its calculations, it must convert these intensities into a format suitable for the display device. If OpenGL is running on a palettized device, it converts each number to a 3-3-2 formatted, 8-bit value.

To see how the process of quantizing to an 8-bit value leads to the mismatch of colors, let's quantize the RGB value (160, 160, 164):

```
(int)160*7/255 = 4 or 100 binary,
(int)160*7/255 = 4 or 100 binary,
(int)164*3/255 = 2 or  01 binary,
```

The quantized value is (4, 4, 2) or (100, 100, 01). Together, these numbers build the 8-bit value 01100100 binary, or 100 decimal. The color at index 100 is (146, 146, 85), which is the value that OpenGL tried to display for us.

Let's apply the same process to the other two colors.

RGB (166, 202, 240):

```
(int)166*7/255 = 4 or 100
(int)202*7/255 = 5 or 101
(int)240*3/255 = 2 or 10
```

The color in index 10101100 binary, or 172 decimal, is (146, 182, 170).

RGB (255, 251, 240):

```
(int)255*7/255 = 7 or 111
(int)251*7/255 = 6 or 110
(int)240*3/255 = 2 or 10
```

The color in index 10110111 binary, or 183 decimal, is (255, 219, 170).

It is interesting that 251 quantizes to 6 instead of 7, although it is very close to 255.

Palette Messages

To be truly palette-aware, an application should handle and support the WM_QUERYNEWPALETTE and WM_PALETTECHANGED messages. A lot has already been written about these functions, including some wonderful words by Nigel Thompson in his book *Animation Techniques for Win32*. GLpal and GLEasy copy the code listed in Chapter 4 of Nigel's book. Amazingly enough, it works.

UpdateColors

You can do things a little differently from Nigel. Instead of invalidating the window and redrawing as GLpal does, you can use **CDC::UpdateColors** to directly update the visible pixels to match the current system palette. In some cases, **UpdateColors** will be faster than redrawing the whole display. (This is especially

true of some complex OpenGL renderings.) However, **UpdateColors** can lead to a loss of information, so you should redraw the display after you use this function a few times. Ron Gery's article "The Palette Manager: How and Why" in the MSDN Library explains **UpdateColors** in detail.

Conclusion

Palettes have always been a pain. Understanding why you need a certain type of palette for OpenGL may initially be a little confusing, but once you have the code to set the palette up correctly, you never need to worry about it again.

By now, you should be familiar enough with OpenGL and palettes to copy the code out of GENGL or GLEasy, and get on with some cool 3-D renderings.

Bibliography

Sources of Information on OpenGL

Crain, Dennis. "Windows NT OpenGL: Getting Started." April 1994. (MSDN Library, Technical Articles)

Neider, Jackie, Tom Davis, and Mason Woo. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Release 1*. Reading, MA: Addison-Wesley, 1993. ISBN 0-201-63274-8. (This book is also known as the "Red Book".)

OpenGL Architecture Review Board. *OpenGL Reference Manual: The Official Reference Document for OpenGL, Release 1*. Reading, MA: Addison-Wesley, 1992. ISBN 0-201-63276-4. (This book is also known as the "Blue Book".)

Prosise, Jeff. "Advanced 3-D Graphics for Windows NT 3.5: Introducing the OpenGL Interface, Part I." *Microsoft Systems Journal* 9 (October 1994). (MSDN Library Archive Edition, Books and Periodicals)

Prosise, Jeff. "Advanced 3-D Graphics for Windows NT 3.5: The OpenGL Interface, Part II." *Microsoft Systems Journal* 9 (November 1994). (MSDN Library Archive Edition, Books and Periodicals)

Prosise, Jeff. "Understanding Modelview Transformations in OpenGL for Windows NT." *Microsoft Systems Journal* 10 (February 1995).

Rogerson, Dale. "OpenGL I: Quick Start.". December 1994. (MSDN Library, Technical Articles)

Rogerson, Dale. "OpenGL III: Building an OpenGL C++ Class." January 1995. (MSDN Library, Technical Articles)

Rogerson, Dale. "OpenGL IV: Color Index Mode." January 1995. (MSDN Library, Technical Articles)

Rogerson, Dale. "OpenGL V: Translating Windows DIBs." February 1995. (MSDN Library, Technical Articles)

Rogerson, Dale. "OpenGL VI: Rendering on DIBs with PFD_DRAW_TO_BITMAP." April 1995. (MSDN Library, Technical Articles)

Rogerson, Dale. "OpenGL VII: Scratching the Surface of Texture Mapping." May 1995. (MSDN Library, Technical Articles)

Microsoft Win32 Software Development Kit (SDK) for Windows NT 3.5 *OpenGL Programmer's Reference*.

Sources of Information on Palettes

Gery, Ron. "Palette Awareness." April 1992. (MSDN Library, Technical Articles)

Gery, Ron. "The Palette Manager: How and Why." March 1992. (MSDN Library, Technical Articles)

Gery, Ron. "Using DIBs with Palettes." March 1992. (MSDN Library, Technical Articles)

Rodent, Herman. "Animation in Win32." February 1994. (MSDN Library, Technical Articles)

Rodent, Herman. "Animation in Windows." April 1993. (MSDN Library, Technical Articles)

Thompson, Nigel. *Animation Techniques for Win32*. Redmond, WA: Microsoft Press, 1995.

Sources of Information on 3-D Graphics

Foley, James D., Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice*. 2d ed. Reading, MA: Addison-Wesley, 1990. (This is THE book on computer graphics.)

Hill, F.S. Jr. *Computer Graphics*. Macmillan Publishing Company, 1990.

Sources of Information on Gamma Correction

Burger, Peter and Duncan Gillies. *Interactive Computer Graphics: Functional, Procedural, and Device-Level Methods.* Reading, MA: Addison-Wesley, 1989.

Rogers, David F. *Procedural Elements for Computer Graphics.* New York, NY: McGraw-Hill, 1985.