Using Color in a Window System: X versus Microsoft Windows

Theo Pavlidis
SUNY at Stony Brook
theo@sbcs.sunysb.edu

Kevin Hunter
NeoMedia Technologies
khunter@neom.com

Contents

Abstract

This is a tutorial paper explaining how to use color under the constraints of a window system for three types of applications: (a) those using a few distinct colors only for labeling or decoration; (b) those using an intermediate number of distinct colors for drawing or economical image display (within a web browser for example); (c) those requiring high color fidelity for the evaluation of image processing or graphics algorithms. The paper also discusses how to minimize the distortion in the appearance of other applications when applicatioons of the third type take control of the color resources of the display.

Each of these topics is discussed in parallel for the X Window System and for Microsoft Windows to help readers who are interested in porting applications from one system to the other.

1. Introduction

Modern graphics displays usually have three 8-bit D/A converters that control the three basic colors (red, green, and blue) so that are able to display over 16 million (2^{24}) different distinct colors. In order to take full advantage of this ability we need a frame buffer (refresh memory) that has 24 bits per pixel. While the price of such display devices keeps decreasing, they are far from being in universal use and 8 bit per pixel devices are still quite common. Such devices can display

Submitted to IEEE CGNA magazine, January 9, 1998.

only 256 colors at any given time and the selection of which particular colors to display is determined by the contents of the video look up table that maps bit patterns into combinations of red, green, and blue (RGB) values. Although 256 colors are inadequate to display arbitrary full-color (i.e. photographic) images, halftoning techniques [Pa96] allow visually pleasing results. Thus, 8-bit systems have historically been an effective trade-off between cost and visual fidelity.

We shall use the term *color table* to refer to the relationship between bit patterns and RGB values. We also distinguish between *logical* and *physical* color tables. The former are part of a program, the latter is what is actually in the hardware. Color tables are important not only for 8-bit systems, but also for systems with greater depth in the case of color overlays. For example, in a 24-bit systems we may have a four part overlay (6 planes each for background and for three moving objects), so that we need to provide the impression of full color with 6 bits.

In the days before graphical user interfaces (GUI's) became common, management of color information was completely device-specific. There was not much difficulty, however, in managing contention for access to the physical color table, since systems typically ran only one application at a time. The running application therefore had complete control over the color table. With the advent of general-purpose GUI systems such as Microsoft Windows and the X Window System, standardized API access to color table facilities was designed. Once displays could support multiple simultaneous applications, however, contention for entries in the color table became an issue.

In this paper we describe how to select the desired colors under both the X Window System (X) and under Microsoft Windows (MW), as well as how multiple "color-aware" applications contend for access to the color table. We review the two systems side by side to help readers who are interested in porting applications from one system to the other. The two systems use both a different design approach and different terminology. For example, X uses the term *colormap* for the color table, while Microsoft Windows use the term *palette* to describe the same thing.

A major overall difference between the two systems is that, under MW, the application(s) and display are always on the same computer, while under X the application and display may be on separate systems. In particular, X includes the concept of a discrete, simple, inexpensive "X Terminal," separate from the computer running applications, that handles all the direct display operations. X allows a client application to display on multiple display servers, and for multiple client applications on multiple computers to place output on a single display server. Thus, MW can enforce a great deal more cooperation across applications sharing a display than can X.

In order to simplify the design of X servers, the color table (colormap) is a responsibility of the application (client). This increases the complexity of the task, but also allows complete control of the details by the application. In contrast, in MW the system handles much of the work. This greatly simplifies the programming of applications with simple color needs, but complicates the programming for applications that need either non-standard or very precise color handling.

Color handling is much easier in systems with 24 bits per pixel (or more) since it is possible to handle each color independently by devoting 8 bits for each of three primary colors. Such systems are normally called *True Color* (in both X and MW). Color handling is a challenge in systems where the mapping of bit-patterns of pixels into the primary colors is programmable because different applications may have different needs and the hardware allows only one mapping at a time. These are typically 8 bit per pixel systems and are called *PseudoColor* in X and *paletted* in MW. We discuss these and other categories in more detail in Section 7 but all other parts of the paper focus on systems with a programmable color table.

We assume that readers have some familiarity with windows programming in the system of interest to them and have already written at least some simple applications using either the *X Toolkit* and *Xlib* in X or the *Win32* API in MW. In order to keep this paper within reasonable

length we limit the discussion to color manipulation functions only and refer the reader to the literature for general drawing functions as well as for further discussion of color ([Ny92] or [SG92] for X and [Ge92], [Th94], [Pe96], or [DL97] for MW).

We also do not discuss color handling in the *Java* Abstract Window Toolkit (AWT) in this paper. In the current AWT implementation, Java windows are implemented in terms of *peer objects* which are windows of the underlying system, X or MW. These windows, thus, use the default color policies of the native system and, as a result, the appearance of Java applications is not consistent across systems. Thus, at least at present, it is difficult or impossible to write serious imaging or graphics applications in "pure Java."

2. Overview of Color Handling in the Two Systems

Let us use the term *color table* (colormap in X or palette in MW) for an array of color entries mapping bit patterns into RGB colors. Applications allocate or construct *logical color tables* containing the colors they wish to use. These logical color tables do not become active, however, until their contents are loaded into the *physical*, or *hardware* color table. When designing a system in which multiple applications will use color simultaneously, the major issue is who will control the contents of the hardware color table. X and MW follow different approaches.

X allows any logical color table to be loaded into the hardware. The system also maintains a *default color table* common to all applications that do not provide their own color table. An application may interrogate the default color table to determine the colors presently there, and may request new entries to be made in the default color table. Because the default color table is a shared resource, when a new applications starts the contents of that table depend on what other applications are running. The window manager is always the first application invoked, and thus has the first opportunity to populate the color table. This means that, at minimum, the colors used for window and menu decorations will be present. Beyond this, however there is no guarantee that any particular color will be in the default color table. Neither is there any guarantee that there will be room in the default color map for new colors, since it possible that running applications have completely populated it.

MW maintains a single system logical color table. This is the only table that is ever loaded in the hardware, unlike X, in which application-specific color tables may be directly loaded. Each application under MW is allowed to have one or more logical color tables associated with specific windows. These logical color tables are used to build the single system color table, under rules enforced by MW. In addition, MW guarantees a specific pre-determined set of *static colors* is always present in the color table. Although technically driver-specific, a set of 20 specific static colors has evolved as a standard that virtually all drivers implement. This insures proper behavior of applications that do not provide or use color tables.

These different designs have major implications in the writing of applications and affect both modest and intensive users of color. In particular, an X application that needs a small number of decorative colors but does not want to use the resource mechanism for all of them (see Sections 3.1 and 4.1 for reasons) must deal with colormap functions right away.

In contrast, a MW application does not have to worry about palette functions as long as it limits its colors to the 20 system colors or is willing to accept approximate color renderings. It does not even have to know what the static colors are exactly; it may specify an RGB triplet and the system will select the static color that is closest to the request (for lines), or select a dithered pattern of colors to approximate the request (for area fills).

An X application that needs a large number of colors (for image display for example) is faced with a hard choice. It must either limit the colors to what can be fit in the default colormap, or it must create its own color table that will be loaded by the window manager into the hardware

when the application gets the focus. X manuals discourage the latter solution because the appearance of the other applications is distorted when a new colormap is loaded.

A MW application that needs a large number of colors may create its own color table (palette) without excessive concern about the appearance of other applications because this color table is never directly loaded into the hardware. Instead the system uses the entries in this table to compose the system color table. Such a composition is not static. The colors of the window on the top of the Z-order take priority over those of windows behind it. Thus, the color table composition is recalculated any time the Z-order changes. Since, in MW, the window with the focus is virtually always on top of the Z-order (unlike in X), the window with the focus almost always has the highest color priority. At the same time the distortions in the appearance of other windows are minimized through the use of color approximations.

A second fundamental difference between MW and X concerns color fidelity. In X, color precision is assured, because a precise color is either available or it is not. If the precise color requested is not in the color map, and no room is available to add it, an error is returned. Under these circumstances MW, on the other hand, will return an index of the closest available color. This has the advantage of providing "best available" color fidelity to an application, even when it is in the background, but makes it difficult for an application to determine the precise color with which it is drawing.

It will be helpful for the subsequent discussion to distinguish three possible uses of color in an application: (a) for labeling or decorations (Section 3); (b) for displaying images for decorative purposes where color fidelity is not a high priority (Section 4); and (c) for displaying images with special attention to color fidelity (Section 5). A special case of (c) is gray scale images in medical and industrial applications where fidelity to the original intensities is important. Finally, we discuss techniques for keeping a consistent appearance of all applications (Section 6) and portability issues (Section 7).

3. Simple Use of Color for Labeling and Decoration

When color is used for labeling or decorations, the exact RGB values are frequently not important. Quite often the programmer would be happy to specify the color by name. X maintains a large database (many hundreds of entries) of color names with respective RGB values, and the X resource mechanism provides for color specification by name. MW does not allow arbitrary color specification by name. Under MW, colors are specified by explicit RGB values. However RGB specification is substantially simpler in MW than in X.

3.1 Using Colors by Name in X

The simplest case in X occurs when you use a single color to draw on window and your program is using one of the widget sets such as Motif or Athena. Then you can take advantage of the resource mechanism and, for example, specify red as the color of the text on all menu buttons in Motif by the line

```
*XmPushButton.foreground:red
```

You may also specify "navy blue" as the color for drawing on the window of the widget named `tablet` by the line

```
*tablet.foreground:navyblue
```

Before assigning a particular color using this mechanism, you should be careful to check for the presence of the color name in the data base. For example, to find all defined variants of "blue", run

```
showrgb | grep blue
```

The collection of available color names is not at all intuitive. For example, in the system used in the department of one of the authors (TP) you might be surprised to find that "dark blue" does not exist, although "dark slate blue", "medium blue", and even "alice blue" do. This is the whimsical side of X!

You can use color names anywhere the color of an object is specified through the resource mechanism. This is typically the case with the color of the text for menu buttons, the background color of windows, etc. On the other hand if you wish to produce a diagram where lines are drawn in different colors such as that shown in Figure 1, the situation becomes more complex.

Figure 1 ("gr.gif") should be here

Figure 1

There are three possibilities. First, you may use the `foreground` resource of the drawing widget (for example, of class `DrawingArea` in Motif) and change its value within the program by a statement such as

```
XtVaSetValues(w, XtVaTypedArg,
    XtNforeground, XtRString, "green", strlen("green")+1, NULL);
```

Even if you are willing to live with the cumbersome syntax, you must face the problem that any time a resource changes value, the X Toolkit causes the widget to be redrawn. This can be annoying if we wish to add an extra line in an existing diagram.

A second solution that avoids the redrawing is the introduction of *application resources*. Here is a code that does that for the example of Figure 1.

```
static Pixel ccolor[4];

static XtResource crayons[] = {
    {"color1", "Color", XtRPixel, sizeof(Pixel),
    (Cardinal)0,
    XtRString, "green" },

    {"color2", "Color", XtRPixel, sizeof(Pixel),
    (Cardinal)(sizeof(Pixel)),
    XtRString, "blue" },

    /* ... etc ... */
};
/* ... */
    XtVaGetApplicationResources(toplevel, /* top level widget */
        (XtPointer) ccolor,
        crayons, XtNumber(crayons), NULL);
/* ...*/
```

The color names in the listing are default values. They may be changed at execution time by command line arguments such as

```
-xrm "*color2:pink"
```

To use a particular color within a program we need only the statement

```
XSetForeground(Dpy, gc, ccolor[i]);
```

where `Dpy` is the display pointer and `gc` the graphics context.

The third way bypasses the resource mechanism entirely, but must deal directly with color tables. We discuss this method in Section 4.1. because most of the material is related to image display.

3.2 Simple Use of Color in MW

MW does not use the resource mechanism, instead, it defines "color schemes" that specify the colors of various entities such as the window backgrounds, the text of menu buttons, etc. Color schemes are specified at the system level by the user, using a Control Panel applet. Under Microsoft Windows 3.1 and Microsoft Windows NT 3.51, this applet was named ‘‘Colors.’’ Under Microsoft Windows 95 and Microsoft Windows NT 4.0, this functionality is on the ‘‘Appearance’’ tab of the ‘‘Display’’ applet. Programs have no direct control over the default colors used for buttons, menus, etc.

MW implements the scheme colors by reprogramming some of the static colors placed in the system color table. MW applications reference scheme colors "functionally." MW provides predetermined functional synonyms for the scheme colors such as `COLOR_WINDOW`, `COLOR_MENUTEXT` and `COLOR_SCROLLBAR`. Thus, a "compliant" application would use a `COLOR_WINDOW` brush to fill the background of its window, rather than a specific RGB value. This ensures that the background of the window is filled with the ‘‘scheme’’ color specified by the user.

Note that, in contrast to X, MW does not provide a mechanism whereby an application user can select individual colors for such things as individual buttons. MW *applications*, however, can be written to override the default color schemes, or provide the user with X-like color control. To to this, the application must take over the drawing of those menus, buttons, etc. for which the colors are to be changed. An application does this by defining the controls as "owner drawn". When controls are so defined, MW notifies the application any time part of the control needs to be drawn. This approach gives the application a great deal of control over the appearance of controls, at the cost of a substantial increase in programming complexity.

For the simple case of the application illustrated in Figure 1 we need very little code. MW does not provide automatic color name conversion, so we must do our own, but in our case this is fairly simple. MW specifies colors using a 32-bit integer known as a `COLORREF`. We can obtain a `COLORREF` value by invoking the macro

```
COLORREF color_value;
color_value = RGB(r,g,b);
```

This macro takes the 8-bit red, green and blue values, and constructs a 32-bit integer. The most significant byte of this integer is zero, and the other three bytes contain the blue, green and red values, in that order. (Note that, unlike X, MW has no provisions for hardware with more than 8 bits per color separation.) In typical applications, the macro is simply used in-stream, as shown below:

```
HPEN hPen = CreatePen(style, width, RGB(r,g,b));
```

When the pen is created, the system finds an approximation (if not an exact match) of the RGB values from among the 20 static colors in the system color table and passes the appropriate index to the brush structure. The particular index to the color table is hidden from the applications programmer. If a brush were created as shown here,

```
HBRUSH hBrush = CreateSolidBrush(RGB(r,g,b));
```

the system will automatically select either a solid color that closely approximates the requested color, or build a dithered brush from the available static colors. Thus, despite the function name, the brush returned by a call to `CreateSolidBrush()` might, in fact, be dithered. Pens, however, are always solid.

In order to draw with a particular color we need to select the appropriate object:

```
SelectObject(hdc, hPen);
```

for line drawings (`hdc` is the device context) or

```
SelectObject(hdc, hBrush);
```

for filled areas. Notice that MW does not have the concept of universal foreground color that X does. MW uses this distinction to allow the programmer to draw, for example, a polygon filled with one color, and outlined with another, in one call, where two calls would be required in X - one to fill and another to outline. Indeed, to outline without filling, for example, a ‘‘null’’ brush must be selected under MW. Outlining without filling, or filling without outlining are common enough operations under MW that the system provides both a null brush and null pen, which may be obtained via

```
HPEN hNullPen = GetStockObject(NULL_PEN);
HBRUSH hNullBrush = GetStockObject(NULL_BRUSH);
```

In this application, there is no requirement for any color table processing, since we do not care about the precise color used to draw the lines. On the other hand, the actual colors may be quite far from what we asked. This is because the approximation to the RGB values is obtained from amongst the 20 static colors reserved by the system. If we ask for `RGB(255, 0 , 0)` it is likely that we will get what we want, but not if we ask for `RGB(170, 149, 48)`. The discrepancy is more likely to occur if the color is used to draw lines or curves rather than to fill areas. This is because when MW creates a brush, it will use dithering (halftoning) to approximate the requested color on the basis of the static set of 20, whereas pens are always solid colors. We can achieve tighter control by specifying our own color table, or palette in the MW terminology. Since this approach is most commonly used with image display it is discussed in Section 4.2.

The resource mechanism under X is powerful and effective because programs and controls under X are designed to load their colors from the resource files, and because these files are easily edited by users to achieve the desired color selections. Through the use of ‘‘owner-drawn’’ controls, a MW application can provide its user with the same degree of control, if desired. Although it is possible for such an application to dedicate a specific portion of its user interface to the selection of colors, an alternate mechanism, paralleling that of the X resource file, is available.

MW supports the concept of a ‘‘profile file’’ containing application settings intended to be persistent across program invocations. These profiles are specially-formatted text files, and MW provides a set of API functions to manipulate them.

Suppose that an application is coded to read from a profile file named `app.ini` and it that the profile file contains the following:

```
[TreeColor]
LeafR=100
LeafG=255
LeafB=0
```

When it is necessary to use the particular color specified by the user, the values can be retrieved as follows:

```
int red = GetPrivateProfileInt("TreeColor", "LeafR", 0,
    "APP.INI");
int green = GetPrivateProfileInt("TreeColor", "LeafG", 255,
    "APP.INI");
int blue = GetPrivateProfileInt("TreeColor", "LeafB", 128,
    "APP.INI");
```

and then used to build the appropriate `COLORREF` as part of the drawing operation.

The first argument to the `GetPrivateProfileInt()` function is a ‘‘section name’’, matching a string contained in brackets. The second argument is a ‘‘key name’’ matching a string to the left of an equal sign. Different sections may use the same keys. The third argument is a default value if no match for the previous strings is found or the file is absent, and the fourth is the name of the

profile file. A similar function named `GetPrivateProfileString()` is available to retrieve non-numeric values, and `WritePrivateProfileString()` is available to programmatically update a file. (There is no `WritePrivateProfileInt()` - the integer must be converted to a string and output using `WritePrivateProfileString()`).

By default, these functions look for the profile file in the Windows directory. It is strongly preferred, however, that an application place this file elsewhere, and use a full path name to specify the location of the file. One common way to do this is to have the profile file in the same directory as the executable. This allows the executable to retrieve its own full path name via a call to `GetModuleFileName()` and then modify the returned string by, for example, replacing the trailing ‘‘.EXE’’ with ‘‘.INI’’. If a full path name must be given explicitly you should use double backslashes, as in `"C:\USERS\MYSTUFF\APP.INI"` since the directory separator in MS-DOS is the escape character in C.

Current Microsoft standards suggest that personal settings such as these be stored in the *registry*, rather than in profile files. This approach has the advantage of allowing different users to have different personal settings, but requires that the application provide a user interface to establish and modify these settings, since the registry cannot be easily edited by users. The fact that profile files can be created and modified using any conventional text editor makes them very useful, if slightly less flexible, alternative.

4. Intermediate Displays of Color

In this section, we discuss applications that require a large number of colors but for which exact color fidelity is not needed. There are two major categories of such applications: drawings with more colors than can be handled by the methods of Section 3 and, most frequently, display of images for decorative purposes.

We have assumed throughout this paper an 8 bit display, so we assume here an 8 bit image as input. This may be a gray scale image or a halftoned color image. Most image files contain both pixel values and color table information and, in order to display an image, the appropriate color entries must be made in the system color table. If the image is going to be used strictly for decorative purposes then colors may be used in an *economical* way. However, if the image is going to be processed, for example by subjecting it to contrast enhancement, and we are interested in comparing the original with the processed result we have to use colors with high fidelity. We discuss the latter case in Section 5 and we focus here on image display where exact color replication is not essential. This is the case with images displayed on the web and displays by such popular utilities as `xv` in X and the `MSPAINT` or `PBRUSH` programs in MW.

The operational definition of such applications is that they require the use of a color table, but their demands are modest enough that there is no need to create a special color table for them.

4.1 Color Facilities in X

The basic X structure for dealing with color in X is the *color cell* defined as

```
typedef struct {
    unsigned long pixel;
    unsigned short red, green, blue;
    char flags;    /* whether to ignore some of the colors */
    char pad;      /* to provide an even number of bytes */
} XColor;
```

Color information is passed between client and server through pointers to such structures. Notice that there are provisions for 16 bits per color separation and for a 32 bit pixel. The pixel value is the one that must be stored in a refresh memory to cause the particular color to be displayed. The bits of the member `flags` are used to determine whether some of the color values passed should

be ignored, an example of X's flexibility at the expense of increased complexity. Usually we want to pass the symbolic value `DoRed | DoGreen | DoBlue`.

In order to create a display we want to provide the RGB values and obtain a pixel value. (Of course we have to get the RGB values from the color name, if we are specifying color by name, but we will worry about that later.) To do that we need to know the color table of the application. This can be found from the widget resources by the following code:

```
Colormap cmap;
Widget w;
XtVaGetValues(w, XtNcolormap, &amp;cmap, NULL);
```

We may also access the default colormap by the following Xlib code

```
Display  *Dpy;
Colormap cmap = DefaultColormap(Dpy, DefaultScreen ( Dpy ) );
```

if we are writing the application without using the X Toolkit (not a recommended practice).

A subtlety of X is that while the application always specifies the three color components,

it does not always specify the pixel value.

If a shared colormap is used (for example the one obtained from the widget resources) and we want to add to it a particular color with given RGB values, we place these values into an `XColor` structure and call the function

```
Boolean XAllocColor(Display *Dpy, Colormap cmap, XColor *ccell_ptr)
```

This call searches the color table for a match with the given RGB values. If an *exact* match is found, its index is returned in the `pixel` member of the XColor structure. If no match is found, an attempt is made to create a new entry in the color table. If this attempt succeeds, the index of its location is returned in the `pixel` member. Thus, in either case, upon successful return, the `pixel` member of the `XColor` structure contains the index to the color table of the desired color. If the specified color is not already in the color table, and there is no room to add the color, the function signifies failure by returning FALSE.

We are left now with the problem of connecting color names to RGB values. There is a function `XParseColor()` that reads the color name data base and extracts RGB values from the names. The following code fragment does the job (error checking has been omitted for brevity).

```
/* Dpy is display pointer, cmap is colormap, and gc is graphics context */
static char *color_names[] = { "green", "brown", "blue", "red" }
Xcolor cells[4];

for (i = 0; i < 4; i++) {
    XParseColor(Dpy, cmap, color_names[i], &cells[i]);
    XAllocColor(Dpy, cmap, &cells[i]);
}
/* To draw with green */
XSetForeground(Dpy, gc, cells[0].pixel);
/* ... */
```

An attempt to add a new color is likely to fail if a color demanding application (for example a web browser) is running at the same time. Because the RGB values already in use by the other application may not be exactly the same as those we are requesting, we might be unable to allocate even a single new color, even though the colors we are requesting are visually indistinguishable (or nearly so) from the colors already in use. Clearly, this is absurd, since in this case we are not concerned with precise color values. As we shall see in Section 4.2 in MW the system automatically picks an approximate color, but in X our application must do this work itself. Color

approximation under X is discussed in Section 4.3.

4.2 Drawing in Color and Displaying Images Under MW

Under MW, if we need to use colors other than the 20 static system colors, we must create a logical color table (palette) for the application. Creation of such tables is discouraged in X, but not in MW. The reason is that in MW the system accommodates the application color table with its own, minimizing color distortion in the rest of the desktop. (We will return on this issue in Section 6.) A palette entry (the counterpart of `XColor` in X) is defined as:

```
typedef struct tagPALETTEENTRY
{
    BYTE peRed;
    BYTE peGreen;
    BYTE peBlue;
    BYTE peFlags;
} PALETTEENTRY;
```

For typical applications, the `peFlags` member of the `PALETTEENTRY` structure is set to zero. An application builds a palette using the following structure:

```
typedef struct tagLOGPALETTE
{
    WORD          palVersion;
    WORD          palNumEntries;
    PALETTEENTRY  palPalEntry[1];
} LOGPALETTE, *PLOGPALETTE, *LPLOGPALETTE;
```

The `LOGPALETTE` structure, or logical palette, is intended as the header to a variable-length block of memory containing information related to the color table. Thus, an application will typically create it as follows:

```
LOGPALETTE *pPalette = (LOGPALETTE *)malloc(sizeof(LOGPALETTE) +
                        (numEntries-1)*sizeof(PALETTEENTRY));
pPalette->palNumEntries = numEntries;
```

The `palVersion` field is always set to 0x0300, as this system of palette management was first introduced in Microsoft Windows 3.0. The `palNumEntries` field indicates the number of active entries in the palette.

The `LOGPALETTE` structure is not the actual palette or color table. Once the logical palette structure has been created and populated, the actual MW palette object is created using the following function:

```
HPALETTE CreatePalette(const LOGPALETTE *);
```

This function returns a "handle" to an internal structure that contains the information from the `LOGPALETTE`, plus additional information maintained by MW. This handle is used to refer to the palette object until it is no longer needed, at which time the function `DeleteObject()` is used to destroy it. Note that the `LOGPALETTE` structure need not be maintained after `CreatePalette()` is called.

It should be observed that the order in which colors appear in the `LOGPALETTE` structure is significant to the process. Colors are processed in the order in which they appear. As a result, if the number of unique colors exceeds the available space in the system palette, colors earlier in the logical palette structure will get preferential treatment.

If a palette is available the drawing color should not be selected with the `RGB()` macro. Rather, one of the following macros should be used:

```
PALETTERGB(r,g,b)
PALETTEINDEX(n)
```

The PALETTERGB macro generates a 32-bit integer in the same format as the RGB macro, but with the most significant byte set to 1. The PALETTEINDEX macro generates a 32-bit integer with the most significant byte set to 2, and with the least significant 16 bits containing the specified index value. (The reason more than 8 bits are stored has to do with "false paletting" support on TrueColor displays and will be covered in Section 7).

The functions that use COLORREF (such as CreateSolidBrush()) look at the most significant byte and interpret the data accordingly. When an application specifies a color using the PALETTEINDEX macro, MW automatically determines the intended color by looking it up in the currently-active application palette. If the PALETTERGB macro is used, the color is determined by finding the closest match to the specified RGB values, however in this case the colors in the application palette are used, rather than the 20 static system colors. In either case, this color is *then* matched to the closest color in the entire system color map, whether the particular color was added by this application or not. No dithering is performed for brushes created using this macro because the resulting color is far more likely to be close to what was asked for than when the RGB macro was used. Note the RGB macro always maps the colors to one of the 20 static colors provided by the system *even if a palette is specified by the application*.

In order to actually use the palette, the application must install it during any drawing operations. The code for these operations is given below. In order to ensure that the GDI understands the colors to be used, a palette must be "selected" and "realized" during the painting process. This is accomplished using the following code:

```
HPALETTE hMyPal;         /* return value from CreatePalette() */
HPALETTE hOldPal;        /* temporary value */
HDC hDC;                 /* Handle to Device Context */
. . .
case WM_PAINT:
    hDC = BeginPaint(hWnd, &amp;ps);
    hOldPal = SelectPalette(hDC, hPalCurrent, TRUE);
    RealizePalette(hDC);
    /* Paint the client area. */
    SelectPalette(hDC, hOldPal, TRUE);
    RealizePalette(hDC);
    EndPaint (hWnd, &amp;ps);
    break;
```

The call to SelectPalette() specifies the palette that should be used for subsequent drawing operations. RealizePalette() performs much of the actual work associated with mapping colors from the selected application palette to the current system palette. Although there were technical reasons in the past for separating this operation into two functions, at present they should always be called as a pair.

SelectPalette() returns a handle to the palette previously installed in the device context. In the case shown, this will always be the system palette. It is an error to leave an application palette selected when a device context is destroyed (in this case, by EndPaint). Thus, the system palette handle is saved in the first call to SelectPalette(), and selected back in after all the drawing is done, displacing the application palette.

In addition to installing the palette during the drawing process, an MW application must inform the system of its desired palette any time the window Z-order changes. This is covered in Section 6.2.

This may be the right point to comment on some of the major differences between X and MW.

In X, an application may attempt to add new colors in the common color table by calls to XAllocColor(). This is not possible in MW because the system color table is not directly accessible by the applications. Thus

an application that is not satisfied with the 20 static system colors must do significantly more work in MW than in X.

However, there are some advantages in keeping the system color table beyond of the reach of the applications. When using the `PALETTEINDEX` macro, the application programmer does not need to be concerned about potential color remappings that take place. The index value specified is an index into the application's palette, not into the system palette. If the color at this entry is mapped to some other color in the system palette, this transformation still takes place during the second translation mentioned above.

4.3 Color Approximation and Economical Image Display in X

While in X an application may create as many color tables as it wants, this practice is strongly discouraged because it will cause distortions in the appearance of all other applications. Therefore applications are encouraged to use a common color table (usually the default colormap) as much as possible, using approximate colors if need be. The price of the simple server design in X is that the management of color conflicts must be done by the application. In contrast, in MW color approximation is done by the system and applications are not discouraged from creating their private color tables, since the system has the responsibility of managing color conflicts.

The following function finds the nearest color in a given colormap to RGB values of a color cell. It stores that value in the color cell. The function uses a Manhattan distance in the color space because it requires less computation than a Euclidean distance. The modification of the code for another distance metric is straightforward.

```
Boolean near_color(Display *Dpy, Colormap cmap, XColor *cp)
{
    long i, d, d0, i0;
    static XColor    existing_cells[256];
    Boolean status = False;

    /*    Find out the existing colors in the given colormap */
    for (i = 0; i < 256; i++) existing_cells[i].pixel = i;
    XQueryColors(Dpy, cmap, existing_cells, 256);

    d0 = 65535*3;    /* distance between pure white and pure black */
    for (i = 0; i < 256; i++) {
        d = abs(cp->red - existing_cells[i].red) +
            abs(cp->green - existing_cells[i].green) +
            abs(cp->blue - existing_cells[i].blue);
        if(d < d0) { d0 = d;  i0 = i; status = True; }
    }
    if(status) cp->pixel = i0;
    return status;
}
```

The `XQueryColors()` function call finds the values of all colormap entries for an 8 bit display. Then we search for the nearest RGB triplet to the given one and store the pixel value in the color cell. This function could be used in the code of Section 4.1 as shown below:

```
for (i = 0; i < 4; i++) {
    XParseColor(Dpy, cmap, color_names[i], cells+i);
    if( XAllocColor(Dpy, cmap, cells+i) == False) {
        /* print warning about using approximate value */
        near_color(Dpy, cmap, cells+i);
    }
}
```

The first step in color economizing during image display is to take advantage of the fact that quite often not all entries in the color table that accompanies the image file are used in an image. It is thus advisable to construct a histogram of the pixel values and identify the values actually in use. Only the colors associated with these values need be added to the default colormap or approximated by existing colors there.

Let `j` be a pixel value and `red[j]`, `green[j]`, and `blue[j]`, be the corresponding color values. For image display, instead of obtaining the RGB values from the database through the `XParseColor()` function, as we did in Section 4.1, we use the image color table entries. The main difference is that the `XColor` structure uses 16 bits per color separation, while images usually have only 8 bits per color separation. As such, we must shift the image values to obtain the corresponding `XColor` values. The following code performs the conversion and color map update:

```
long remap_pixel[256];
XColor tmp;

/* [calculation of image histogram omitted] */

for (j = 0; j < 256; j++) {
if (the jth entry in the histogram is non-zero)
    {
    tmp.red   = red[j] << 8;
    tmp.green = green[j] << 8;
    tmp.blue  = blue[j] << 8;
    if( XAllocColor(Dpy, cmap, &tmp) == False) {
        /* no room - find nearest color */
        near_color(Dpy, cmap, &tmp);
    }
    remap_pixel[j] = tmp.pixel;
    }
}
```

The next task is to update the pixel array using the information in the pixel remapping array. This can be done with code such as

```
for (y = 0; y < image_height; y++)
    for (x = 0; x < image_width; x++)
        pixel[y][x] = remap_pixel[ pixel[y][x] ];
```

In order to display the image we must create an `XImage` structure, but this is straightforward now that the pixel array values have been remapped. The `data` member of this structure need only point to the updated `pixel` array. Of course, having modified the `pixel` array as shown, we have lost our original colors, so any thought of image processing using the `pixel` array is now out of question. In image processing applications, it would be necessary to save the approximated pixels into a separate array to avoid this. Image processing applications, however, are unlikely to find the approximate color display adequate, and thus will more likely use the techniques of Section 5.

The code above allows the display of an image with as much fidelity as is possible given the available room in the color map. The code could easily be enhanced by considering the most frequently used colors first, so that heavily-used colors were more likely to be rendered exactly than rarely-used colors.

Thus, except for the larger number of colors used, display of images for decorative purposes is based on the same techniques as drawing with color.

5. Image Display with High Color Fidelity

There are several reasons that the economical approach for image display may be undesirable. When approximate colors are used, different color values in the original may be mapped to the same displayed color, so that the image shows unsightly contours. In addition, approximate colors are unacceptable in cases where we study the effects of image processing. In the case of X, the default colormap may not have enough space, therefore we must often create a new colormap. However, there are cases where we might be able to use the default colormap and we discuss this case first. In MW the key idea is to force the system to use the colors of the application palette at the expense of other colors.

When an application uses a private colormap in X, it normally asks the window manager to load the colormap in the hardware whenever the application has the focus. An MW application's palette is automatically given priority by the system whenever it is on top of the Z-order. While, under these conditions, such an application is shown in its exact colors, other applications are likely to be shown in distorted colors because the bits in the refresh memory are not changed, but the mapping between these bits and displayed colors does change. We discuss in Section 6 ways to minimize such distortions.

5.1 High Color Fidelity Image Display Under X Without a New Colormap

It is likely that serious image processing work on 8 bit displays will be limited to *gray scale* or single color images. Because the human eye has difficulty discriminating intensity values differing by less than 1%, it is often acceptable to ignore the least significant bit in each pixel, thus effectively reducing our problem to 7 bit images. In this case we might be able to obtain the colors from the default colormap. We may avoid the indirection implemented by the `remap_pixel[]` array in the economic display by asking for contiguous color cells. In particular, rather than asking for pixels one at a time, we ask for a block of them using the function

```
Boolean XAllocColorCells(Display *Dpy, Colormap cmap,
                         Boolean contiguous,
                         unsigned long plane_masks[], int nplanes,
                         unsigned long pixels[], int RGB_length)
```

By setting the third argument to `True`, we ensure that obtain a contiguous array of color cells. The function offers the option of allocating either planes or color cells. The former is useful for creating color overlays but since we are not interested in that for the time being, the values of `nplanes` should be 0. `RGB_length` is the number we color cells we want, 128 in the case of a 7 bit image.

In order to add the colors in the color table we must create an array of color cells where the `pixel` value is *copied* from the array `pixels[]` and the color values from the image color table. The `red[]`, `green[]`, and `blue[]` arrays are usually part of the image data, or they may be implicit. For example, for an 8-bit gray scale image where each pixel value, `i`, represents its intensity we may construct the color table for the 7 bit version by the code:

```
for (i = 0; i < 128; i++)
    red[i] = green[i] = blue[i] = i<<1;
```

The color cell array is constructed by the following code:

```
XColor ccell[256];

for (i = 0; i < RGB_length; i++) {
    ccell[i].pixel = pixels[i];         /* obtained from XAllocColorCells() */
    ccell[i].red   = red[i]<<8;         /* associated with the image */
    ccell[i].green = green[i]<<8;       /* associated with the image */
    ccell[i].blue  = blue[i]<<8;        /* associated with the image */
```

```
        ccell[i].flags = DoRed | DoGreen | DoBlue;
    }
```

After the `XColor` array is formed, it is loaded in the color table with the following function:

```
Boolean XStoreColors(Display *Dpy, Colormap cmap,
                     XColor *ccell, int RGB_length)
```

This is similar to `XAllocColor()` except that the pixel values of the color cells are *defined* in this case, rather than being *returned*, as with `XAllocColor()`.

There is one more task. A pixel with intensity `i` is represented by entry `pixels[i]` in the color table. The first pixel index in the allocated group, corresponding to intensity zero, is almost certainly not zero, so again we must remap the pixels to conform to the color table. However, since the pixel values are contiguous we only need to add a fixed value, `pixels[0]` to all image pixels. There is a *Xlib* function that does that:

```
XImage *xi;

XAddPixel(xi, pixels[0]);
```

Because the value of the images have only a constant added to them, unlike the more radical remapping of the economical display algorithm, one might be able to use the data of the `XImage` structure for some image processing operations.

The approach we just described is likely to fail if we try to run two copies of the application because there will not be enough space in the default colormap for two copies of the 128-entry RGB arrays. In this case, the second application's call to `XAllocColorCells()` will fail. This can be corrected by having the application respond to this failure by reading the current system color map using `XQueryColors()`, determine if the desired block of entries has already been placed there and, if so, use them.

5.2 High Color Fidelity Image Display Under X Using a Private Colormap

The option of last resort for high color fidelity display under X is to create a private color map. This gives the application complete control of its appearance, at the expense of introducing (probably significant) distortions in the desktop and other applications. A new colormap is created by a call such

```
Colormap new_cmap = XCreateColormap( Dpy, DefaultRootWindow(Dpy),
    DefaultVisual(Dpy, DefaultScreen(Dpy)), AllocAll);
```

The meaning of the first two arguments is obvious. The third argument refers to a `Visual`, the place where X keeps information about the hardware of the video look up table (see Section 7.1). The fourth and last argument is a symbolic constant that specifies that the application wants to use all the color cells. (The other allowable value is `AllocNone` in which case cells will by allocated by calls to `XAllocColors()` or `XAllocColorCells()`.)

To populate the newly created colormap, we construct a color cell array as in Section 5.1 and then place it by `XStoreColors()`. If possible, it is advisable for applications to avoid placing custom colors into all entries of the colormap. The first few entries in the default colormap are usually the decorative colors installed by the window manager. If these colors are copied to the corresponding position in the application color map, distortions in the appearance of other applications will be greatly reduced. We can do that by calling `XQueryColors()` (see Section 4.3) for the default colormap and copying entries to the new colormap. (Copying 10-12 entries is usally sufficient.) See Section 5.3 for the comparable situation under MW.

Applications should never explicitly load the colormap. This task is left to the window manager, which is responsible for automatically loading the colormap whenever the application has the focus. However the application must inform the window manager about the new map through the

call.

```
XtVaSetValues(wtop, XtNcolormap, new_cmap, NULL);
```

where `wtop` is a shell widget, normally the top widget of the application.

It is possible to have an application with different colormaps for different subwindows, although some servers may not handle the situation correctly. The following code shows an example. The widgets that take a private colormap are placed in an array and the window manager is informed about that array through the top widget. Each widget may have different colormaps, even though in the example are all the same.

```
Colormap new_cmap;  /* private colormap */
Widget wtop;        /* top widget of the application */
Widget w[N];        /* widgets that take the new colormap */

for (i = 0; i < N; i++)
    XtVaSetValues(w[i], XtNcolormap, new_cmap, NULL);
XtSetWMColormapWindows(wtop, w, N);
```

X encourages the use of *standard colormaps* whenever there is a need for a private colormap. This feature is not particularly useful, however, because when a private colormap is required, its colors are usually derived from an image, and thus are unlikely to match any of the existing "standards." While it is possible for an application to create its own standard colormap, the code to do so correctly is quite subtle. Furthermore, the only real benefit of the use of standard colormaps is to decrease the number of colormaps that must be maintained by the server, thus conserving memory. While worth this was significant in the days of scarce and expensive memory, it is not generally a concern today.

5.3 High Color Fidelity Image Display under MW

The key idea in high color fidelity displays in MW is to keep the system informed of the colors you want to use by passing palette information to it. Palette information is needed both when you draw (WM_PAINT), as well as when the Z-order of the window changes. The need in the former case is obvious. The need in the latter case results from the system’s process of allocating colors based on windows’ Z-order.

Under normal circumstances, MW will allow an application to specify up to 236 custom colors, since it reserves 20 static colors for itself. It is possible for an application to override 18 of the 20 static colors, thus allowing up to 254 custom colors. This is done via the function call

```
SetSystemPaletteUse(hdc, SYSPAL_NOSTATIC);
```

where `hdc` is the handle of the device context and `SYSPAL_NOSTATIC` a symbolic constant. In this case, the system reserves only two static colors: pure black and pure white. The call should be made only when the window is maximized and has the input focus. There are several more actions that an application must take after such a drastic change, as well as to restore the system to its normal condition, but this topic is beyond the scope of this tutorial, and is discouraged in practice.

The corresponding situation in X was discussed in Section 5.2. It is probably worthwhile commenting on the design differences between X and MW as well as the underlying philosophy. In both systems it is possible for an application to take almost all the color table entries. In X this is the default option and the application is responsible for leaving a few "crumbs" to other application by calling `XQueryColors()` and preserving some of the old colors in the new color table. In MW the default is to leave the static colors alone and a special (and considerable) effort is required to take over them, and even then two colors are left undisturbed. Thus, in this case, the design philosophy of X was to allow the application the maximum degree of control available.

MW, on the other hand, restricts the control of an application in the interests of overall system appearance.

It was mentioned at the beginning of this section that an application must provide MW with palette information whenever the Z order changes. When an application window is moved to the top of the Z order, MW sends it a WM_QUERYNEWPALETTE message. This affords this application the chance to place its colors into the system palette. Whenever the system palette is changed, *all* applications in the system are sent the WM_PALETTECHANGED message. The combination of this pair of messages enables the dynamic palette negotiation between applications. An application would typically respond to these messages as follows:

```
case WM_PALETTECHANGED:
    if (wParam == hWnd)         /* Responding to own message. */
        break;                  /* Nothing to do. */
    /* otherwise drop through to next case */
case WM_QUERYNEWPALETTE:
    hDC = GetDC(hWnd);

    /* select and realize our palette */
    hOldPal = SelectPalette(hDC, hMyPal, FALSE);
    iTemp = RealizePalette(hDC);

    /* put back original palette */
    SelectPalette(hDC, hOldPal, TRUE);
    RealizePalette(hDC);

    ReleaseDC(hWnd, hDC);
    return(iTemp);
```

This sequence of code responds to the WM_QUERYNEWPALETTE message by selecting and rendering the application palette, then reselecting and rendering the original system palette. This sequence of code both informs the system that this is a palette-aware application and provides its desired palette. When a WM_PALETTECHANGED message is received, the same sequence is executed. The extra test in the WM_PALETTECHANGED case is to prevent a potential infinite regress and is discussed below.

Understanding the operation of this code involves understanding both the third parameter (bForceBackground) to SelectPalette() and how the system uses it. If the current application is the foreground application (i.e. is on the top of the Z-order) and SelectPalette() is called with bForceBackground set to FALSE, all entries in the system palette (except for the static colors) are cleared before the application palette is applied. If the current application is not the foreground application or bForceBackground is TRUE, the current colors in the system palette are left untouched, and the colors in the application palette added.

Thus, when the current foreground application receives a WM_QUERYNEWPALETTE message and executes this code, it first selects its own palette with bForceBackground set to FALSE. This causes all the non-static colors in the palette to be flushed, and as many as possible of the colors in hMyPal to be placed into the system palette. Following this, the palette that was originally in the DC is replaced. Note that, in this case, bForceBackground is set to TRUE, to prevent this operation from undoing the work just completed. When the application responds to a WM_PALETTECHANGED message originated by another application, the same sequence is executed. Since the application is not the foreground one, system behaves as if bForceBackground were TRUE, regardless of the value passed in, and the colors are merely added to the table. Thus, an application can never override colors placed into the system palette by an application that is on top of it in the Z order.

The test in the WM_PALETTECHANGED case is necessary to prevent an infinite regress. Whenever

the system palette changes, all applications *even the application causing the change* are sent a `WM_PALETTECHANGED` message. Without this test, the code above would respond to the `WM_PALETTECHANGED` message by calling `SelectPalette()` with `bForceBackground` set to `FALSE` when it is the active application. If this were to happen, the palette would be re-initialized by `SelectPalette()` and `RealizePalette()`, causing another `WM_PALETTECHANGED` to be generated, yielding an infinite loop. This could also be avoided by coding the `WM_PALETTECHANGED` case separately with `bForceBackground` set to `TRUE`, however this results in unnecessary extra processing by the system.

An application needing to display a bitmap can do so readily by using the palette associated with the bitmap in the manner shown above. The Microsoft Device Independent Bitmap (DIB or BMP) format carries with it a palette in essentially the same form required by the `LOGPALETTE` structure. This makes it relatively simple to display a single DIB. If two or more DIB's need to be displayed, the easiest approach is to assign a priority order for the bitmaps. Palettes are then created from each DIB, and successively selected and realized in response to the `WM_QUERYNEWPALETTE` and `WM_PALETTECHANGED` messages. The first palette is selected with `bForceBackground` set to `FALSE`, and all others with `TRUE`. Finally, during the paint operation, each palette is selected and realized just before the corresponding bitmap is placed on the screen. DIB's earlier in the priority order will get better color fidelity, just as applications higher in the Z order get better color fidelity, but the results are usually acceptable, if not perfect.

Throughout this discussion we have mentioned the role of the foreground application. What happens if the foreground application does not use palettes? Who, if anyone, gets the `WM_QUERYNEWPALETTE` in this case? The Palette Manager in Windows version 3.1 and later sends the `WM_QUERYNEWPALETTE` message to the topmost palette-using application in the Z-order if the new foreground application does not use palettes. If no palette-using applications are currently running and the desktop is drawn with a bitmap, the desktop is given palette priority. A palette-using application in this case is defined as an application that has at some point explicitly called `SelectPalette()`.

Finally, before terminating, the terminating application only needs to worry about deleting any palette objects it created. The system handles the rest.

6. Keeping a Consistent Desktop Appearance

We discuss here how to minimize the distortions in the appearance of application when an application with a private color table takes priority by the methods discussed in Section 5.

In MW, the system attempts to minimize the distortion of other applications by using the static colors for decorations, and providing nearest-color matching for other colors. Thus an application need take no action if it is not particular about its appearance when is not on the top of the Z order. A modest amount of code may improve on the appearance provided by the system and this is discussed in Section 6.2.

In X, no action is taken by the system to maintain the appearance of other applications and decoration colors are not static. Thus, when a private colormap is loaded by one application, most other applications typically appear quite distorted unless the application that created the private colormap saved some of the colors of the default colormap in the way discussed in Section 5.2. This solution may not be convenient for such programs as image browsers or image processing programs where we want to present as faithful a representation of the colors as possible. Such browsers may contain slider bars for a viewport and/or menu buttons, all taken from a widget set such as Motif. While we may ignore the appearance of other applications we would like to maintain (at least approximately) the colors of the widgets of the image browser.

It is possible to do so by a process that is complementary to the economical image display described in Section 4.3. Instead of approximating the image colors, we approximate the control widget colors. The method is described in Section 6.1. It is important that the private colormap be assigned only to the image window and the Window Manager be informed about it through the `XtSetWMColormapWindows()` function as described in Section 5.2.

6.1 Keeping a Consistent Application Appearance in X

The key idea in maintaining a consistent application appearance under X is to use `XColor` structures instead of `Pixel`s to keep track of color resources. The RGB values of the `XColor` are used to find approximate pixel values as colormaps change. X generates an `ColormapNotify` event when the colormap of a window changes, and places information about the nature of the change in the `xcolormap` member structure of the `XEvent` union. Therefore we need to register an event handler for `ColormapNotify` (the selection mask is `ColormapChangeMask`) and respond to this notification accordingly.

The `ColormapNotify` event is generated under one of three conditions that are listed below together with the respective values of members of the `xcolormap` structure.

* When a new colormap has been assigned to a window. Then `xcolormap.new` is `True`
* When a window's private colormap is loaded into the hardware. Then `xcolormap.new` is `False` and `xcolormap.state` takes the symbolic value `ColormapInstalled`.
* When a window's private color map is unloaded from the hardware and replaced by another color map. Then `False` and `xcolormap.state` takes the symbolic value `ColormapUninstalled`.

It is important to note that a window receives a `ColormapNotify` event only when *its own* colormap is involved. Thus, suppose there are three windows A, B and C which monitor this event. Further, suppose that A and B overlap so that it is possible for the cursor to move directly from A to B. When focus passes from A to B, A receives a `ColormapUninstalled` indication, and B receives a `ColormapInstalled` indication. Window C, however, receives no notification, since its colormap was neither loaded nor unloaded. Thus, there are limits to the ability of an X application to track and adapt to colormap changes.

There are other problems. If an application changes colors while running it will have to keep track of whether its own colormap is loaded or not. If, however, we limit our attention to widget color resources, then the situation is more tractable.

When the window is first realized an event of the first type is generated and at that time we may find the pixel value of the resource and subsequently fill its RGB colors by the following code:

```
Widget w;
XColor z;

XtVaGetValues(w, resource_name, &(z.pixel), NULL);
XQueryColor(XtDisplay(w), colormap, &z);
```

When the colormap is unloaded an event of the third type is generated and we must try to match or approximate the RGB values in the new colormap. Unfortunately, in this case, the `xcolormap` structure contains no information about the contents of the newly loaded colormap. Although there is an `xcolormap.colormap` member, this always contains the identity of the target window's own colormap, not the one being loaded. To retrieve the colormap that is loaded we must call the function `XListInstalledColormaps()` which returns a pointer to a list of the installed colormaps. (It is possible for more than one colormap to be installed, however this is an unlikely case.) Let `new_map` be the installed colormap and `near_color()` the color approximating function of Section 4.3. Then the calls

```
near_color(xtDisply(w), new_map, &z);
XtVaSetValues(w, resource_name, z.pixel, NULL);
```

will produce a new value for `z.pixel` and update the resource value. When a resource value changes the X Toolkit also redraws the widget, thus guaranteeing the appearance of the new colors.

When the original colormap is re-installed (event of the second type) we could repeat the procedure we used with events of the third type but it takes less processing to save the original pixel values for each resource when the windows are created and retrieve them now.

Thus, in summary, the technique operates as follows:

* When a widget is created, it loads its color using the resource mechanism, saving the corresponding pixel value.
* When the `ColormapNotify` event is received indicating the assignment of a new colormap to the window, the pixel value is retrieved, and translated to the corresponding RGB triplet using `XQueryColor()`. This triplet is saved.
* When another application's colormap is loaded, that colormap is retrieved, and `near_color()` is used to find the pixel value closest to the RGB triplet. This pixel is passed to the widget, which indirectly forces a repaint in the new approximate color.
* When the window's own colormap is loaded, the original pixel value is restored, again forcing the widget to repaint.

Finally, we must decide where to keep the `XColor` structure and, possibly, the original pixel values. The simplest solution is to pass them as client data to the colormap event handler. A sample code implementing such an arrangement can be found in the Appendix. The function `FixMotifColors()` takes as its argument the top level widget of the application, and must be called just before `XtRealizeWidget()`.

6.2 Optimizing Background Application Appearance in MW

In Section 5.3 we discussed drawing with color under MW, but not how to optimize the display fidelity when the application is not the active application. When some other application changes the palette, so that the pixel-to-color translations have changed, our application has three choices:

First, we can do nothing. In this case, the displayed colors in our application may be radically altered as a result of changes in the system palette. This may be acceptable. However, if we take this approach and a portion of our window is uncovered while we are not the foreground application, the `SelectPalette() / RealizePalette()` operation in the `WM_PAINT` processing will choose colors from the system palette that best approximate the colors we request. The areas of our window that were *not* repainted, however, will still have the original bits in the refresh memory. It is quite likely that these bits now correspond to colors which are very different from those intended. As a result, our window will take on a "patchwork" appearance, in which different portions of the window will be painted with different colors, depending on the nature of the system palette at the time that portion of the window was last repainted. This is different from the behavior under X. In X, the system does not perform the color approximation. Thus, a background X application redrawing an exposed portion of the window draws with the original bit patterns, and thus the same distorted colors, avoiding the "patchwork" appearance.

Second, we could choose to repaint the entire window any time the palette changes. This option results in the best color fidelity, at the expense of extra processing. We can cut down on the number of extra repaints by taking advantage of the return value of `RealizePalette()`. This return value indicates the number of entries in the system palette that changed as a result of the realization process. If this value is zero, then the colors have not actually been altered. This can

happen when there is only one application using a palette, or when the total number of colors being requested fits with the system palette. If `RealizePalette()` returns a non-zero value, we can use `InvalidateRect()` to force a repaint operation on the window.

Finally, we can adapt to the new colors without repainting. Microsoft Windows provides a function called `UpdateColors()` that can update all the visible screen pixels in a device context to match the current system palette as closely as possible. This approach has the advantage of speed, since the translation is handled by the operating system, but will gradually cause the colors to "drift", since they can undergo a succession of "nearest neighbor" matches.

A compromise solution is to use a combination of full repainting and adaptation. The first few times the palette changes, UpdateColors is called. Once a certain number of calls to `UpdateColors()` have taken place, a full repaint occurs. This is illustrated in the following code:

```
    case WM_QUERYNEWPALETTE: /* only received when we're in foreground */
        hDC = GetDC(hWnd);
        hOldPal = SelectPalette(hDC, hPalCurrent, FALSE);
        i = RealizePalette(hDC);     /* Realize drawing palette. */
        SelectPalette(hDC, hOldPal, TRUE);
        RealizePalette(hDC);
        ReleaseDC(hWnd, hDC);
        if (i != 0)                  /* if the realization changed */
        {
            InvalidateRect(hWnd, NULL, TRUE);    /* force a repaint  */
            gUpdateCount = 0;        /* Starting update tracking  */
                                    /* from scratch. */
        }
        return(i);
    case WM_PALETTECHANGED:
        if (wParam != hMyWnd)
        {
            hDC = GetDC(hWnd);
            hOldPal = SelectPalette(hDC, hPalCurrent, TRUE);
            /* Only need to repaint if logical palette is remapped. */
            if (RealizePalette(hDC) != 0)     /* Realize drawing palette. */
            {
                /* If fewer than two updates have been done, we can  */
                /* update. */
                if (++gUpdateCount < 2)
                    UpdateColors(hDC);
                /* Otherwise, it's time to repaint from scratch. */
                else
                {
                    gUpdateCount = 0;
                    InvalidateRect(hWnd, NULL, TRUE);
                }
            }
            SelectPalette(hDC, hOldPal, TRUE);
            RealizePalette(hDC);
            ReleaseDC(hWnd, hDC);
        }
        break;
    case WM_PAINT:
        if (gUpdateCount > 0)        /* If painting after some  */
                                    /* updating... */
        {
            BeginPaint(hWnd, &amp;ps);
            EndPaint(hWnd, &amp;ps);
```

```
        gUpdateCount = 0;        /* Starting update tracking from  */
                                 /* scratch. */
        InvalidateRect(hWnd, (LPRECT) NULL, TRUE);
        break;
    }
    else
        /* Usual paint stuff. */
    break;
```

Note the call to `InvalidateRect()` in the `WM_QUERYNEWPALETTE` case. Under normal circumstances, MW clips GDI operations to that portion of its window that is newly exposed as it comes to the foreground. This means that if a window were partially exposed, and the exposed portion was processed with `UpdateColors()` while the window was in the background, that portion of the window would contain the wrong pixel values when the window came to the foreground and the application's palette was re-installed, yet would not be repainted by the `WM_PAINT` processing. Thus, if this might have happened, the application must force a full repaint via `InvalidateRect()` when it comes to the foreground to ensure proper color display. Since an application only receives the `WM_QUERYNEWPALETTE` message when it is the topmost palette-using application, this is an ideal time to do this. Performing this call only when `RealizePalette()` returns non-zero, indicating that the color mapping has changed, eliminates unnecessary extra repaints.

Similarly, if a `WM_PAINT` message is received, and `gUpdateCount` is non-zero, the application calls `BeginPaint()` and `EndPaint()` to handle this message, but then calls `InvalidateRect()` to invalidate the entire window. This will result in another `WM_PAINT` message being generated immediately, with the clipping region set to the entire window. This ensures that the entire exposed portion of the window is draw with a consistent set of colors.

Finally, note that when the image to be displayed is in shades of gray, rather than color, an elegant technique is available for maximizing its fidelity. The key to this technique is to store the gray values in the palette in non-linear order. For example, suppose the palette entries were made in the following order

```
128, (64, 192), (32, 96, 160, 224), (16, 48, 80, 112, 144, 176, 208, 240), ...
```

The values are ordered as in a binary tree, with values in the same "depth" shown within parentheses. When the application selects and realizes this palette, the "coarser" values will be placed into the system palette first, with the finer values following only if there is room. Since the system will automatically remap the image pixel values as the bitmap is painted, no manipulation of the image data is required. Thus, with this approach, no matter how much room is available in the system palette, the result is an evenly-spaced set of gray shades, maximizing the fidelity of the displayed image.

7. Portability between Systems with Different Color Table Implementation

It is always desirable to write applications that so that they run on as many systems as possible. When dealing with color, the challenge is to make provisions for interrogating the display device about its color table facilities and then to select an appropriate subset of the code depending on the answer. The two parts of this section discuss the process for X and MW respectively.

7.1 X Visuals

X servers maintain a structure of type `Visual` that contains information about their color display capabilities. The format of this structure is "opaque", however, and X applications determine the display capabilities from the information structure `XVisualInfo`, obtained by using the function `XGetVisualInfo()` .

Even if you are not worried about portability, you cannot ignore visuals because the Xlib functions that create a colormap or an `XImage` structure need a visual as an argument. However in such cases all you need is a pointer to a visual and you may obtain it through the widget resource mechanism using the following code:

```
Visual    *visual;
Widget    w;
/* ... */
XtVaGetValues(w, XtNvisual, &visual, NULL);
```

A pointer to a visual is also returned by the following macro call.

```
Display    *Dpy;
Visual    *visual;
/* ... */
visual = DefaultVisual( Dpy, DefaultScreen(Dpy) );
```

This is the approach we took in Section 5.2.

Even if you worry about portability, you may frequently avoid dealing with visuals by looking at the depth of the display screen (number of bits per pixel in the refresh memory) instead of the visual information. This information is kept as a widget resource and can accessed with it with the following code:

```
Cardinal    depth;
Widget    w;
/* ... */
XtVaGetValues(w, XtNdepth, &depth, NULL);
```

or by using an Xlib macro:

```
Display    *Dpy;
Cardinal    depth = DefaultDepth(Dpy)
```

For typical graphical user interface application, the available depth may not be critical. For serious graphics and imaging applications, however, it usually is. Such applications may have to use halftone images to display them, or may even be forced to exit if the available depth is too small.

If you are really ambitious about portability, then you must ‘‘bite the bullet’’ and deal with visuals. The first member of the `XVisualInfo` structure is a pointer to a `Visual` structure, while the rest include the number of colormap entries, the number of bits per colormap index, and, most important, a `class` member that refers directly to the capacity and programmability of the video look up table. There are six possible classes, and they are referred to by the symbolic names `DirectColor`, `PseudoColor`, `GrayScale`, `TrueColor`, `StaticColor`, and `StaticGray`. The first three imply the presence of at least one programmable video look up table, while the other three imply fixed table(s).

The classes `DirectColor` and `TrueColor` imply that the refresh memory can accommodate three *separate* indices for the red, green, and blue colors and, of course, three D/A converters so that the full range of colors supported by the device can be displayed at the same time. `TrueColor` is usually found in color displays with 24 bits per pixels, where there is no need for a programmable colormap unless one wishes to create overlays using different pixel planes. `DirectColor` indicates the presence of three separate lookup tables, one each for red, green and blue. This makes it possible to change the mapping between bits and color intensities.

The classes `PseudoColor` and `StaticColor` imply that there is only one index to the colormap but three D/A converters. `PseudoColor` implies a programmable colormap, and is typical of color displays with 8 bits per pixel. `StaticColor` implies that the system can display only a fixed, non-programmable list of colors.

The classes `GrayScale` and `StaticGray` imply that there is only one D/A converter and shades of only one color are shown. (It need not be actually gray since the color seen depends on the phosphorus of the display!) `GrayScale` implies a programmable colormap, and `StaticGray` a fixed one.

Clearly, there is a hierarchy amongst the three types of devices: If a device can support `PseudoColor` it can also support `GrayScale` by setting the values of the three basic colors equal to each other. Similarly, if a device can support `DirectColor` it can also support `PseudoColor` by setting the three color indices equal to each other.

One should be aware that the information provided by visuals is often not reliable for two reasons: (1) Even if the device supports a certain functionality, the server implemententation may not have provided for the proper visual support. (2) At the other extreme, the visual may reflect what the hardware *can do* rather than what it *actually does*. A typical case of the second situation occurs when, on a particular X server, a color monitor is replaced by a monochrome monitor. This is easy to do: we need connect it only to the output of the green D/A converter. There is no way to ensure that the server is aware of such an external wiring change, so interrogating the visual structure will provide the answer that the display supports color, whereas effectively it does not. (At least, not that the user can see!)

A well-written application, then, might have two sets of label colors: one set with real colors and the other with shades of gray. If the visual says that there is no support for color, the program selects the gray shades. If color support is indicated, the color labels are selected. A *very* well-written application would also provide a user-provided flag, environment variable or resource setting to force the use of the gray labels on a display that appeared to support color.

The simplest way to check the visual structure is by using the function `XMatchVisualInfo()`. Its use is illustrated by the following code for the common case where we want to find whether there it is possible to load a colormap for an 8 bit per pixel display.

```
static Display *Dpy;
/* ... */
    Visual *v = find_visual(8, PseudoColor);
/* ... */
Visual *find_visual(int desired_depth, int desired_type)
{
    XVisualInfo vtemp;

    if( XMatchVisualInfo(Dpy, DefaultScreen(Dpy), desired_depth,
        desired_type, &vtemp) ) return vtemp.visual;
    else return (Visual *)0;
}
```

The returned visual should be used as argument to the Xlib functions that require it.

7.2 Writing Applications for Both Paletted and Non-Paletted Devices

Under MW, it is relatively straightforward to write an application that functions properly on both paletted and non-paletted displays.

An application may determine whether the device on which it is drawing is paletted or not by calling the `GetDeviceCaps()` function as follows:

```
if (GetDeviceCaps(hDC, RASTERCAPS) & RC_PALETTE)
```

```
{
    /* paletted device */
}
else
{
    /* non-paletted (TrueColor) device */
}
```

If the application wishes to always behave in a paletted manner with regard to `WM_PAINT` processing, it may. If an application uses `SelectPalette()`, `RealizePalette()` and `PALETTERGB` or `PALETTEINDEX`, a non-paletted display will correctly dereference the colors. As such, if a paletted display will serve the application's needs, it does not need to be re-written to work on a non-paletted display.

If an application wants to take full advantage of a TrueColor display, and detects it as shown above, all it needs to do is to omit the calls to `SelectPalette()` and `RealizePalette()` and to specify its colors using the `RGB` macro instead of using `PALETTERGB` or `PALETTEINDEX`. This generally means only minor changes are required to the `WM_PAINT` processing. One exception to this is the case of drawing bitmaps that themselves use palettes. When such a bitmap is being painted, the palette information is required to render the bitmap properly. In this case, therefore, the creation of a bitmap palette and the calls to `SelectPalette()` and `RealizePalette()` are still required during `WM_PAINT` processing. These calls may be omitted for pen- and brush-based drawing, however, as well as when drawing bitmaps that do not contain palettes (e.g. 16- or 24-bit-per-pixel bitmaps).

On a TrueColor display, an application will never receive the `WM_QUERYNEWPALETTE` or `WM_PALETTECHANGED` messages. As a result, nothing special needs to be done to modify this processing. On 16-bit, 24-bit and 32-bit display devices, MW supports logical palettes of up to 4,096 entries. Thus, in this case, an application that chooses to retain its palette-based code even on a TrueColor device has the option to expand past 256 color palettes. This is occasionally useful.

8. Conclusions

We have discussed the problems associated with using color under the X Window System (X) as well as under Microsoft Windows (MW). The major differences between the two systems are that

* In MW the system mediates the color requests of applications and does not let them have direct access to the hardware color table, while X allows direct access.
* If a requested color is not available, MW will automatically use the closest approximation to that color, remapping pixel values as needed to make this approximation. X either returns the precise requested color or an error, and does no pixel remapping. (X attempts color approximation only in systems without a programmable color table).

As a result, in MW an application may create its own color table without concern about its effects on other applications or a need to remap its own data. The system manages these issues, and high color fidelity is guaranteed when the window of the application is on top of the Z-order.

In contrast, applications in X are discouraged from creating their own color tables and may have to perform data remapping in order to use the common system color table. If they do create their own table, the appearance of other windows is usually distorted. Thus, what appears at first to be a freer environment is actually more restrictive, requiring more programming effort to obtain "proper" application behavior.

In this respect porting color intensive applications from X to MW is an easier task than the other way around. Consider for example 8-bit gray scale images that can be displayed quite adequately with only 7 bits (the effect of the least significant bit being effectively invisible). In MW one needs only a palette with 128 entries of gray. There is no need to modify the data because the system does color approximation, in effect ignoring the least significant bit. X does no approximation, so the applications must do the remapping itself. In addition, pixels must be remapped to compensate for the unpredictable location of the gray values in the colormap.

Further, suppose we have other color intensive applications running, so that there may not be 128 places available in the system colormap. In X the application must deal explicitly with the problem. A foreground application in MW never sees this condition, because the system palette is dynamically managed, with the foreground application having priority. An MW application running in the background may choose to ignore the issue, or may maintain an approximate appearance by simply repainting its window, and relying on MW's color approximation procedures to give it the best representation possible.

On the other hand, the fact that the pixel-to-color relationship is precisely known, and that an application *can* gain complete control under X, allows special effects such as overlays to be handled in the color maps without undue problem. The color approximation procedure under MW, and the resulting arbitrary (and silent) remapping of pixel values, makes such techniques substantially more difficult on MW systems.

Thus, to a certain degree, the design of the X color system versus that of MW parallels the differences between UNIX and Windows. X and UNIX are more powerful and flexible than Windows, but are more difficult to master and use properly. Windows, on the other hand, permits the more mundane tasks to be performed fairly easily, has some inherent limits that can complicate the more sophisticated tasks. With the appropriate diligence, however, an experienced programmer can produce complex, high-fidelity color displays on either system.

References

* [Ge92] R. Gery. "The Palette Manager: How and Why," March 23, 1992. Available on the *Microsoft Developer Network Library CD-ROM* or through *MSDN On-Line* at

* [DL97] P. DiLascia. "More Fun with MFC: DIBs, Palettes, Subclassing, and a Gamut of Reusable Goodies" *Microsoft Systems Journal*, Vol 12 No. 1, Jan 1997.

* [Th94] N. Thompson. *Animation Techniques in Win32*, Microsoft Press, 1994.

* [Ny92] A. Nye. *Xlib Programming Manual*, 3rd ed. Sebastopol, Calif.: O'Reilly & Associates, 1992.

* [Pa96] T. Pavlidis. *Interactive Computer Graphics in X*, PWS Publishing Company, Boston, 1996.

* [Pe96] C. Petzold. *Programming Windows 95* Microsoft Press, 1996.

* [SG92] R. W. Scheifler and J. Gettys. *X Window System*, 3rd ed. Burlington, Mass.: Digital Press, 1992.

Appendix: Sample Code

```
/*   Color Stabilization for Motif Widgets (1/5/98)         */
/*   When another colormap is loaded, the best closest  */
/*   set is found.                          */
/*   See end of file for public function          */

#include <X11/StringDefs.h>
#include <X11/Intrinsic.h>

/*   Widgets with Special Color Resources */
#include <Xm/ScrollBar.h>
#include <Xm/ToggleB.h>
#include <Xm/PushB.h>

#define MX_C  6    /* at most 6 color resources per Motif widget */

typedef struct {
     XColor rgb[MX_C];
     Pixel ocolor[MX_C];
} ColorSaver;

#define INIT_COLOR    0
#define RESTORE_COLOR 1
#define APPOX_COLOR      2

/*   Maintain color stability for a Pixel resource of a widget   */

static void fix_resource(w, cause, csp, cdata, resource, index)
     Widget w;
     ColorSaver *csp;
     XtPointer cdata;
     char *resource;
     int index;
{
     switch(cause) {
     case INIT_COLOR:  /* First Time Colormap is attached to Window */
          XtVaGetValues(w, resource, &(csp->rgb[index].pixel), NULL);
          csp->ocolor[index] = csp->rgb[index].pixel;
          /*   Fill RGB values for pixel   */
          XQueryColor(XtDisplay(w), (Colormap)cdata, &(csp->rgb[index]));
          return;
     case RESTORE_COLOR:    /* Own Colormap (re)installed */
          XtVaSetValues(w, resource, csp->ocolor[index], NULL);
          return;
     case APPOX_COLOR: /* Other Colormap installed */
          if(match(&(csp->rgb[index]), (XColor *)cdata))
               XtVaSetValues(w, resource, csp->rgb[index].pixel, NULL);
          return;
     }
}

/*   Maintain color stability for ALL Pixel resources of a widget    */
static void motif_colors(w, n, csp, cdata)
     Widget w;
     ColorSaver *csp;
     XtPointer cdata;
{
     if(XtIsSubclass(w, xmPrimitiveWidgetClass)==False
```

```
          && XtIsSubclass(w, xmManagerWidgetClass)==False) return;
     fix_resource(w, n, csp, cdata, XtNbackground, 0);
     fix_resource(w, n, csp, cdata, XmNforeground, 1);
     fix_resource(w, n, csp, cdata, XmNtopShadowColor, 2);
     fix_resource(w, n, csp, cdata, XmNbottomShadowColor, 3);
     fix_resource(w, n, csp, cdata, XmNhighlightColor, 4);

     if(XtClass(w)==xmScrollBarWidgetClass) {
          fix_resource(w, n, csp, cdata, XmNtroughColor, 5);
     }
     else if(XtClass(w)==xmToggleButtonWidgetClass) {
          fix_resource(w, n, csp, cdata, XmNselectColor, 5);
     }
     else if(XtClass(w)==xmPushButtonWidgetClass) {
          fix_resource(w, n, csp, cdata, XmNarmColor, 5);
     }


}

/*   Color approximation using Manhattan distance  */
/*   Changes only pixel value, leaves RGB intact   */
static int match(cellp, given)
     XColor *cellp, *given;
{
     long i, d, d0, i0;
     XColor *cp;
     int status = 0;

     d0 = 65535*3; /* distance between pure white and pure black */
     for(i=0, cp=given; i<256; i++, cp++) {
          d = abs(cellp->red - cp->red) +
               abs(cellp->green - cp->green) +
               abs(cellp->blue - cp->blue);
          if(d<d0) { d0 = d;  i0 = i; status = 1; }
     }
     if(status) cellp->pixel = i0;
     return status;
}

/*   Event Handler for Colormap Change    */

static void c_change(w, client_data, ep, disp)
     Widget w;
     XtPointer client_data;
     XEvent *ep;
     Boolean *disp;
{
     ColorSaver *csp = (ColorSaver *)client_data;

     if(ep->xcolormap.new) {
          /*   Establish Colors   */
          motif_colors(w, INIT_COLOR,
                    csp, (XtPointer)ep->xcolormap.colormap);
     }
     else {
          if(ep->xcolormap.state==ColormapInstalled) {
               /* Restore Colors */
               motif_colors(w, RESTORE_COLOR, csp, NULL);
```

```
          }
          else {   /* Own Colormap Uninstalled - Approximate Colors */
               int i;
               int nc;
               XColor existing_cells[256];
               Display *Dpy = XtDisplay(w);
               Colormap *i_cmap = XListInstalledColormaps(Dpy,
                              DefaultRootWindow(Dpy), &nc);
               for(i=0; i<256; i++) existing_cells[i].pixel = i;
               XQueryColors(Dpy, *i_cmap, existing_cells, 256);
               motif_colors(w, APPOX_COLOR,
                         csp, (XtPointer)existing_cells);
          }
     }
}

/*   PUBLIC FUNCTION                              */
/*   Stabilize resource colors of widget tree     */
/*   It should be called just before XtRealizeWidget()  */

/*   The function adds an event handler for colormap changes    */
/*   to each widget in the tree. Space is allocated for */
/*   client data in the event handler that is used for  */
/*   passing information between invocations           */

void FixMotifColors(w)
     Widget w;
{
     int i, nk;
     Widget *children;

     if(XtIsComposite(w)) {
          XtVaGetValues(w, XtNnumChildren, &nk, NULL);
          XtVaGetValues(w, XtNchildren, &children, NULL);
          for(i=0; i<nk; i++) FixMotifColors(children[i]);
     }
     XtAddEventHandler( w, ColormapChangeMask, False, c_change,
               (XtPointer)malloc(sizeof(ColorSaver)) );
     /* we should probably worry about malloc failure */
}
```