Chapter 2

Creating and Controlling Displays

The CMX11 display utilities provide a functional interface to the X Window System from CM Fortran and C* programs. The library is structured so that you can open and control an X display with just a few calls and without direct X programming. However, if you are an experienced X programmer or need to integrate your CMX11 program with an existing X application, you may want to use the routines that allow you to access the underlying X11 structures. See Section 2.2.1 for more information on accessing X identifiers.

Note that many of these CMX11 routines are simple wrappers for X11 routines of the same base name. C* programmers are free to use either set interchangeably.

2.1 Creating a Display

As described in Section 1.4.1, the basic procedure for creating a CMX11 display is to specify the display to be used by setting the **`DISPLAY`** environment variable or by using the **`-display`** command line option, and then to call **`CMXCreateSimpleDisplay`** or **`CMXCreateDisplayClass`**. We will look more closely at what the create routines do, but first we should look at another option that you can exercise before creating the display: you can set the window and icon titles you wish to be displayed.

Copyright © 1993 Thinking Machines Corporation

2.1.1 Setting the Window Title

If the window manager you are running on the display supports window borders, a window title appears as a label in the top border of the window when a CMX11 display is opened. By default the title is "CM Display." If you want a different title, you can call **CMXSetXWindowTitle** before creating the window, and you can determine the default title currently set by calling **CMXGetXWindowTitle**:

From CM Fortran:

```
CALL CMXSetXWindowTitle(window_title)
CHARACTER*(*) window_title

CHARACTER*(*) FUNCTION CMXGetXWindowTitle()
```

From C*:

```
void CMXSetXWindowTitle(char *window_title);

char* CMXGetXWindowTitle(void);
```

Similarly, the CMX11 routines that create a display set an icon name to be used to label the window when it is closed to an icon. By default the icon name is "cmdisplay"; routines are available to set and return this name as well:

From CM Fortran:

```
CALL CMXSetXIconName(icon_name)
CHARACTER*(*) icon_name

CHARACTER*(*) FUNCTION CMXGetXIconName()
```

From C*:

```
void CMXSetXIconName(char *icon_name);

char* CMXGetXIconName(void);
```


Version 1.6, October 1993
Copyright © 1993 Thinking Machines Corporation

NOTE

The values set by **CMXSetXWindowTitle** and **CMXSetXIconName** are read when the window is created. You must call these routines to set the window and icon defaults *before* creating the window or they will have no effect.

2.1.2 Opening a Display

The **CMXCreateSimpleDisplay** or **CMXCreateDisplayClass** routines create a window on the X server specified by the **DISPLAY** environment variable or by the **-display** command line option:

From CM Fortran:

```
INTEGER FUNCTION CMXCreateSimpleDisplay(ncolors, x, y)

INTEGER FUNCTION CMXCreateDisplayClass(class, ncolors, x, y)
```

From C*:

```
CMX_display_t CMXCreateSimpleDisplay(int ncolors,
                                     int x, int y);

CMX_display_t CMXCreateDisplayClass(int class,
                            int ncolors, int x, int y);
```

The *ncolors* argument specifies the number of colors you wish the display to support. The routines use *ncolors* to determine the visual class of the display that they create.

The arguments *x* and *y* are the dimensions of the window you wish to open, in pixels.

When either routine creates a display, it

- opens a connection to the X server specified in the **DISPLAY** environment variable or with the **-display** command line option.

Copyright © 1993 Thinking Machines Corporation

- opens a window on the display screen sized to *x* by *y* pixels unless overridden by the window manager or by user-defined defaults set in the X resources.
- creates a graphics context set to a background color of black and to the other standard X11 defaults. The graphics context defines parameter values that control the way drawing is done in the window, for example, line width, line style, etc. (See Section 3.1.3 for more information on the graphics context.)
- returns a CMX11 display indentifier that can be passed to other CMX11 routines that operate on the display.

The two routines differ in the way they assign a visual class to the display. The visual class defines how color values drawn to the display are to be interpreted and may be any of the standard X classes: PseudoColor, StaticColor, DirectColor, TrueColor, GrayScale, or StaticGray. The visual classes are described in detail in Section 2.3.3.

`CMXCreateSimpleDisplay` matches the number of colors you request in *ncolors* with the capabilities of the physical display to determine an appropriate visual class for the display. **`CMXCreateDisplayClass`** allows you to specify the class directly. The specified class is used to determine which of two equally likely visual classes to use. For example, a display that supports 8 bits per pixel and an RGB colormap may be configured as a PseudoColor or a GrayScale display. With **`CMXCreateDisplayClass`** you can tell the system which to use.

NOTE

If the requested number of colors or the requested visual class is not available, **`CMXCreateSimpleDisplay`** and **`CMXCreateDisplayClass`** connect to the display using the best available visual class; they do not fail or signal an error. You should always use the color information routines described in Section 2.4 to check the color characteristics of the display before you begin to draw to it.

Version 1.6, October 1993
Copyright © 1993 Thinking Machines Corporation

2.2 Information Routines

2.2.1 X Identifiers

Once you have created a display, you will need to get various X identifiers associated with it in order to identify the display to other CMX11 routines. These routines take the CMX11 display identifier returned by **CMXCreateSimpleDisplay** or **CMXCreateDisplayClass** and return integers identifying the underlying X11 data structures. These identifiers, in turn, may be passed on to the CMX11 information and drawing routines to identify the CMX11 display.

The CMX11 library includes information routines to return the parameters you will need to identify the display to the other routines:

To return the X11 **Display** identifier, use **CMXGetDisplay**:

From CM Fortran:

```
INTEGER FUNCTION CMXGetDisplay(cmx_display_t)
INTEGER cmx_display_t
```

From C*:

```
Display* CMXGetDisplay(CMX_display_t cmx_display_t);
```

The **Display** identifier is passed to all CMX11 drawing routines; to the color management routines **CMXAllocateColors**, **CMXStoreColors**, and **CMXStoreColor**; and to the double buffering routine **CMXHasMBuffers**.

To return the X11 **Drawable** identifier, use **CMXGetDrawable**:

From CM Fortran:

```
INTEGER FUNCTION CMXGetDrawable(cmx_display_t)
INTEGER cmx_display_t
```

From C*:

```
Drawable CMXGetDrawable(CMX_display_t cmx_display_t);
```

The **Drawable** identifier is passed to all the CMX11 drawing routines.

Copyright © 1993 Thinking Machines Corporation

To return the **GC** (graphics context) identifier, use **CMXGetGC**:

From CM Fortran:

```
INTEGER FUNCTION CMXGetGC(cmx_display_t)
INTEGER cmx_display_t
```

From C*:

```
GC CMXGetGC(CMX_display_t cmx_display_t);
```

The **GC** identifier is passed to all the CMX11 drawing routines and the color convenience routines **CMXSetForeground** and **CMXSetBackground**.

To return the X11 **Colormap** identifier, use **CMXGetColormap**:

From CM Fortran:

```
INTEGER FUNCTION CMXGetColormap(cmx_display_t)
INTEGER cmx_display_t
```

From C*:

```
Colormap CMXGetColormap(CMX_display_t cmx_display_t);
```

The **Colormap** identifier is passed to the color management routines **CMXAllocateColors**, **CMXStoreColors**, and **CMXStoreColor**.

To return an identifier for the Motif **Widget** that is the drawing-area widget for the CMX display, use **CMXGetWidget**.

From CM Fortran:

```
INTEGER FUNCTION CMXGetWidget(cmx_display_t)
INTEGER cmx_display_t
```

From C*:

```
Widget CMXGetWidget(CMX_display_t cmx_display_t);
```

The drawing-area widget is a data structure that controls the characteristics of the window's drawing surface. The **Widget** identifier is passed to all the information routines that return information about the window and to the double buffering routines **CMXAllocMBuffers**, **CMXSwitchMBuffers**, and **CMXFreeMBuffers**.

Copyright © 1993 Thinking Machines Corporation

2.2.2 Window Size Information

You can check the actual dimensions of a CMX11 display window by calling **CMXGetWidth** and **CMXGetHeight**:

From CM Fortran:

```
INTEGER FUNCTION CMXGetWidth(widget)
INTEGER widget

INTEGER FUNCTION CMXGetHeight(widget)
INTEGER widget
```

From C*:

```
int CMXGetWidth(Widget widget);

int CMXGetHeight(Widget widget);
```

These routines return the current width and height of the drawing area in pixels.

NOTE

Remember that most X window managers allow the user to resize the window on the workstation screen at any time, even as the program is running. Before you draw to the window each time, your program should use the information routines **CMXGetWidth** and **CMXGetHeight** to confirm the size of the window and, if necessary, scale your image appropriately.

2.2.3 Color Information Routines

A complete set of color information routines allow you to determine the color configuration and capabilities of the display:

- **CMXGetDepth**
- **CMXIsColor**

Copyright © 1993 Thinking Machines Corporation

- **CMXIsRGB**
- **CMXHasColormap**
- **CMXColormapSize**
- **CMXBitsOfRed**
- **CMXBitsOfGreen**
- **CMXBitsOfBlue**
- **CMXColormapRedSize**
- **CMXColormapGreenSize**
- **CMXColormapBlueSize**

Each of these routines takes a *widget* argument and returns an integer. These routines are described in detail in Section 2.4, below.

2.3 Display Color

One of the ways in which physical displays vary most widely is in their color characteristics. CMX11 includes a set of routines that allow you to determine the color capabilities of a display and to respond to them by loading in standard CMX11 colormaps or by writing and loading your own maps.

To understand how to use these routines, you must know something about how color is defined on X windows. This section provides a brief introduction to display depth, colormaps, and the X Window System visual classes. If you are already familiar with the X color system, you can skip this section and go on to Section 2.4, which describes the CMX11 routines that control and return information about the display's color.

2.3.1 Colormaps

The color values specified in the drawing routines do not directly determine the color displayed. Rather they are used as an index into the display's colormap. The colormap is a look-up table of color values. Usually, the colormap is made up of color cells, each of which contains three values that define a color as the relative *intensity* of the three primary additive colors, red, green,

Copyright © 1993 Thinking Machines Corporation

and blue (see Figure 1). These intensities correspond to the phosphors, red, green, and blue, that physically create the color at each pixel on color monitors. The values are specified as integers in the range 0 to 65535.

Colors are created on the monitor by "mixing" these three colors according to the intensity specified for each. So, for example, (65535, 0 , 0) is red, (0, 65535, 0) is green, and (0, 0, 65535) is blue. You can produce a shade of gray by assigning equal intensity levels to all three colors, from black (0, 0, 0) to white (65535, 65535, 65535). Other colors are produced by varying the mixture of the colors. Some examples are given in the table below.

Red	Green	Blue	Color
65535	65535	0	Yellow
0	65535	65535	Cyan
65535	0	65535	Magenta
42578	10485	10485	Brown
52428	32768	12451	Gold
52428	12451	12451	Orange
55704	37355	28835	Tan

2.3.2 Display Depth

The *depth* of the display is the number of bits per pixel supported by the display screen hardware. The number of bits available determines the length of the colormap. An 8-bit PseudoColor workstation can support a colormap of 256 entries, thus providing 256 colors at any one time. A PseudoColor workstation of 4 bits per pixel supports a 16-entry colormap, and so on. These bits of color information are mapped to the screen pixels according to the visual class defined below.

2.3.3 Visual Classes

CMX11 uses the standard visual classes supported by X Window Systems: PseudoColor, StaticColor, DirectColor, TrueColor, GrayScale, and StaticGray.

Copyright © 1993 Thinking Machines Corporation

PseudoColor/StaticColor

PseudoColor and *StaticColor* use a single color value as an index into the display colormap. PseudoColor workstations allow the colormap to be changed, while StaticColor maps cannot be changed.

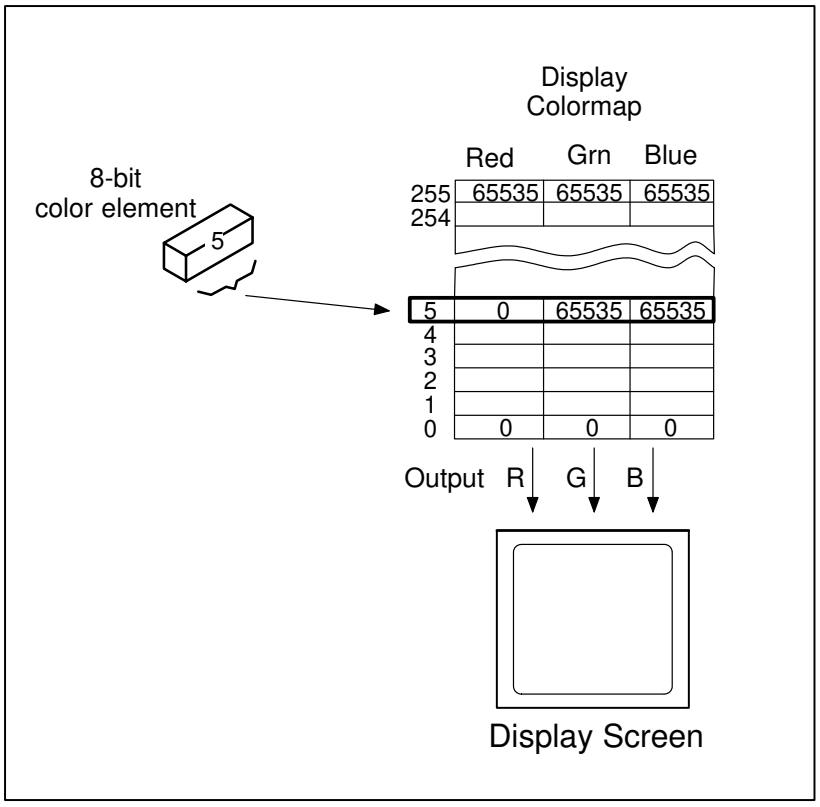

Figure 1. Color mapping of 8-bit color values.

DirectColor/TrueColor

In *DirectColor* or *TrueColor* mode, the color value is interpreted as separate red, green, and blue color intensities. When the color value is written to the display, it is decomposed into three values, each of which indexes a separate component of the RGB colormap. DirectColor workstations allow the colormap to be changed; TrueColor colormaps cannot be changed.

For example, as shown in Figure 2, a 24-bit TrueColor workstation expects 8 bits for each color component. The lowest 8 bits determine the red component, the next 8 bits determine the green component, and the high 8 bits

Copyright © 1993 Thinking Machines Corporation

determine the blue component. In this mode you can vary the intensity of each color component directly, creating much finer gradations of color than with PseudoColor.

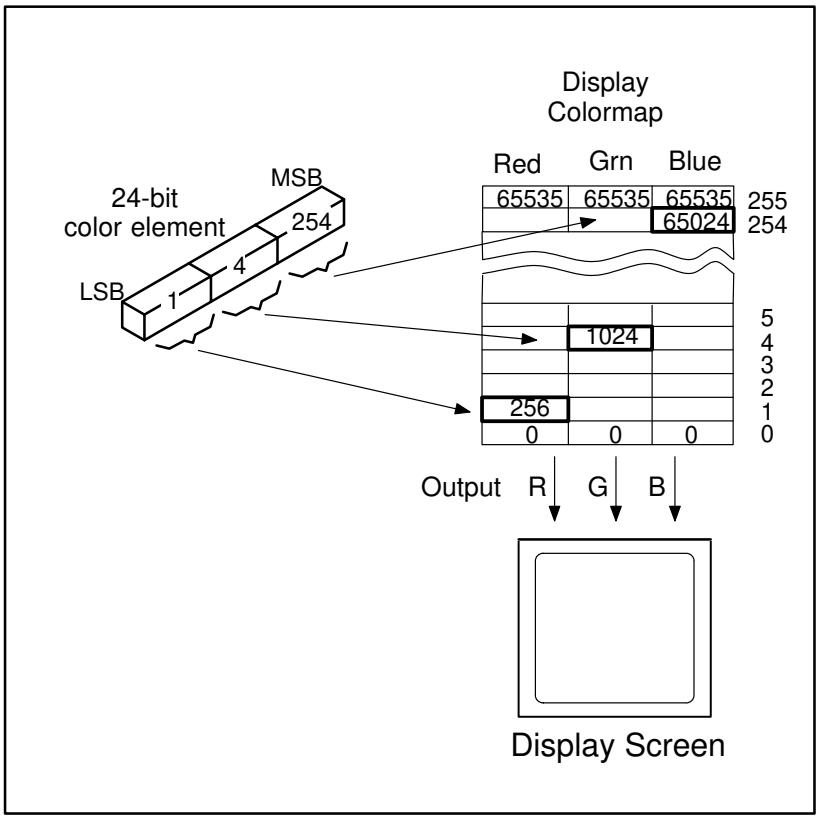

Figure 2. Color mapping of 24-bit color values.

GrayScale/StaticGray: Non-RGB Workstations

GrayScale or *StaticGray* workstations maintain a colormap that contains only a single intensity value at each entry. This value specifies an intensity of gray tone from white to black. GrayScale workstations allow the colormap to be changed; StaticGray colormaps cannot be changed.

The number of bits per pixel determines the number of shades of gray the workstation supports. At the simplest level, a monochrome workstation maintains only 1 bit per pixel, indicating whether the pixel is on or off. An

Copyright © 1993 Thinking Machines Corporation

8-bit GrayScale workstation, as shown in Figure 3, supports 256 gradations of gray.

An RGB colormap can be set to a scale of grays by setting all three values in each cell to the same intensity.

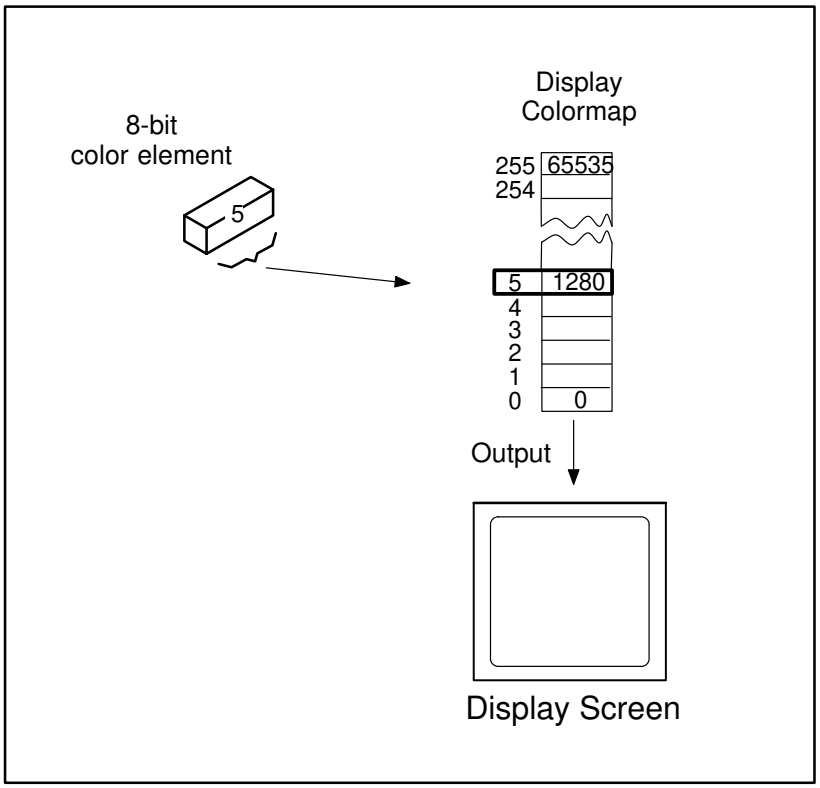

Figure 3. GrayScale color mapping.

2.4 Getting Color Information

2.4.1 Determining the Depth

The most basic piece of information you need about the display is its depth. This is returned by **`CMXGetDepth`**:

Version 1.6, October 1993
Copyright © 1993 Thinking Machines Corporation

From CM Fortran:

```
INTEGER FUNCTION CMXGetDepth(widget)
INTEGER widget
```

From C*:

```
int CMXGetDepth(Widget widget);
```

This is the number of bits of color information maintained per pixel on the currently selected display. This number is also sometimes called the *bits-per-pixel* of the display.

2.4.2 Determining the Visual Class

Once you know the depth of the display, three functions allow you to determine the way the bits of color information are interpreted:

- **`CMXIsColor`** returns TRUE if the display is a color display or FALSE if the display is grayscale or monochrome:

  From CM Fortran:

  ```
  LOGICAL FUNCTION CMXIsColor(widget)
  INTEGER widget
  ```

  From C*:

  ```
  logical CMXIsColor(Widget widget);
  ```

- **`CMXIsRGB`** returns TRUE if the display's pixel values contain separate red, green, and blue components:

  From CM Fortran:

  ```
  LOGICAL FUNCTION CMXIsRGB(widget)
  INTEGER widget
  ```

  From C*:

  ```
  logical CMXIsRGB(Widget widget);
  ```


Copyright © 1993 Thinking Machines Corporation

- **CMXHasColormap** returns TRUE if the display supports a writable colormap:

  From CM Fortran:

  ```
  LOGICAL FUNCTION CMXHasColormap(widget)
  INTEGER widget
  ```

  From C*:

  ```
  logical CMXHasColormap(Widget widget);
  ```

As summarized in the following table, these three routines together allow you to determine the visual class of the display:

Display Type	Result of CMXIsColor	Result of CMXIsRGB	Result of CMXHasColormap
TrueColor	TRUE	TRUE	FALSE
DirectColor	TRUE	TRUE	TRUE
StaticColor	TRUE	FALSE	FALSE
PseudoColor	TRUE	FALSE	TRUE
StaticGray	FALSE	FALSE	FALSE
GrayScale	FALSE	FALSE	TRUE

2.4.3 Describing the Colormap

For displays that do have a colormap, **CMXColormapSize** returns the number of entries in the colormap:

From CM Fortran:

```
INTEGER FUNCTION CMXColormapSize(widget)
INTEGER widget
```

From C*:

```
int CMXColormapSize(Widget widget);
```

This function always returns the size of the largest of the colormaps, red, green, or blue. In most cases the sizes of these maps are the same. However,

Version 1.6, October 1993
Copyright © 1993 Thinking Machines Corporation

some DirectColor or TrueColor displays may have a different length for each color component. For example, an 8-bit display can be used to simulate a DirectColor display by using 3 bits for the red component, 3 bits for the green, and 2 bits for the blue component. In this case the red and green maps would have 8 entries while the blue map would have only 4 entries. For such displays, **`CMXColormapSize`** returns 8. To determine the size of each map in such cases use the following routines; they return the number of bits used for each color component:

From CM Fortran:

```
INTEGER FUNCTION CMXBitsOfRed(widget)
INTEGER widget

INTEGER FUNCTION CMXBitsOfGreen(widget)
INTEGER widget

INTEGER FUNCTION CMXBitsOfBlue(widget)
INTEGER widget
```

From C*:

```
int CMXBitsOfRed(Widget widget);

int CMXBitsOfGreen(Widget widget);

int CMXBitsOfBlue(Widget widget);
```

The sum of the values returned by these routines equals the depth of the display. If the current display is not a TrueColor or DirectColor display, these functions return zero.

Another set of routines return the number of entries in each of the colormaps:

From CM Fortran:

```
INTEGER FUNCTION CMXColormapRedSize(widget)
INTEGER widget

INTEGER FUNCTION CMXColormapGreenSize(widget)
INTEGER widget

INTEGER FUNCTION CMXColormapBlueSize(widget)
INTEGER widget
```


Copyright © 1993 Thinking Machines Corporation

From C*:

```
int CMXColormapRedSize(Widget widget);

int CMXColormapGreenSize(Widget widget);

int CMXColormapBlueSize(Widget widget);
```

2.5 Setting the Colormap

When you create a display with **CMXCreateSimpleDisplay** or **CMXCreateDisplayClass**, you specify in the *ncolors* argument the number of hardware colors the visual should support.

Before you can use the colors you must allocate the cells, create color arrays, and load the arrays into the colormap.

2.5.1 Allocating Color Cells

To allocate the cells in the display colormap, use the CMX11 routine **CMXAllocateColors**:

From CM Fortran:

```
SUBROUTINE CMXAllocateColors(display, cmap, ncolors, base)

INTEGER display
INTEGER cmap
INTEGER ncolors      ! IN/OUT
INTEGER base         ! OUT
```

From C*:

```
void CMXAllocateColors(Display *display,
             Colormap cmap, int *ncolors, int *base);
```

The *display* and *cmap* identifiers passed to these routines are those returned by **CMXGetDisplay** and **CMXGetColormap** respectively. The *ncolors* argument is the number of contiguous color cells you wish to allocate and is modified by the function to contain the actual number allocated. The *base*

Copyright © 1993 Thinking Machines Corporation

argument, as explained below, is set by the function to specify the starting location of your cells in the colormap.

The colormap is shared by all the windows using the display. As illustrated in Figure 4, the cells you request can only be allocated from the unused cells and may begin anywhere in the colormap. On return, **`CMXAllocateColors`** sets *ncolors* from the number of color cells you requested to the number actually allocated and sets the *base* parameter to the first color cell you are allocated in the colormap. To refer to your first color, you index *base*; to refer to the second, you index *base* + 1; etc.

NOTE: It is good programming practice to allocate only the minimum number of color cells you need for your application. This leaves color cells available for other applications and makes your program portable to more X servers.

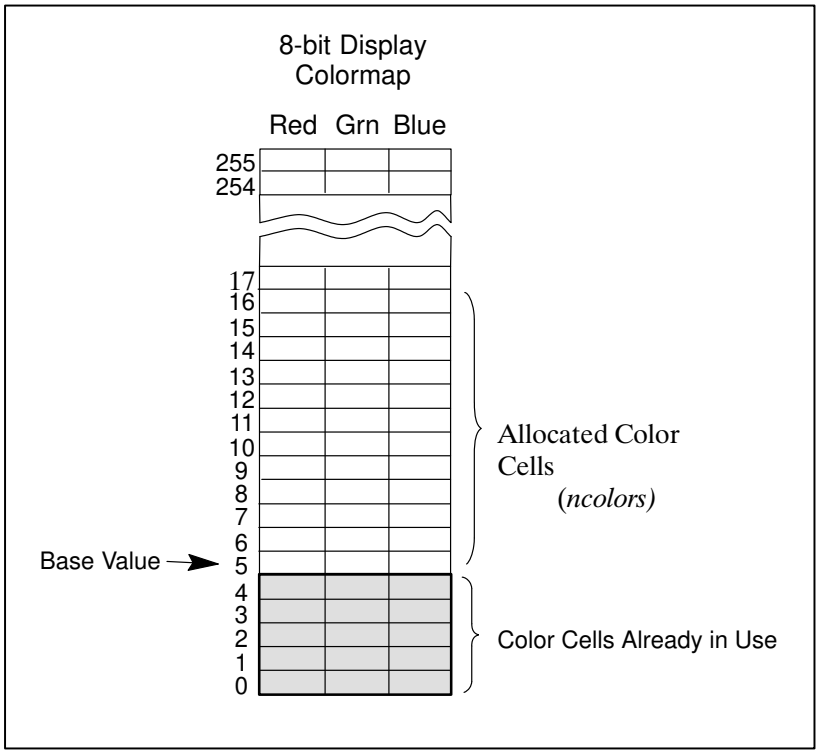

Figure 4. Allocating color cells.

Copyright © 1993 Thinking Machines Corporation

NOTE

If the number of color cells you request in the *ncolors* argument is not available, **CMXAllocateColors** allocates as many color cells as possible to you and sets *ncolors* to the number actually allocated.

You should always check the returned value of *ncolors* to determine how many color cells you may use and adjust your programming appropriately.

2.5.2 Writing to the Color Cells

To set the color values over the entire range of the color cells you have allocated, you can write color arrays with **CMXStoreColors**:

From CM Fortran:

```
SUBROUTINE CMXStoreColors(display, cmap, base, len, r, g, b)

INTEGER display
INTEGER cmap
INTEGER base
INTEGER len
INTEGER r(len), g(len), b(len)
```

From C*:

```
void CMXStoreColors(Display *display, Colormap cmap,
       int base, int len, int r[], int g[], int b[]);
```

The *base* argument is the starting color cell index returned by **CMXAllocateColors**. The *len* argument is the number of elements in the color arrays and should be no longer than the value of *ncolors* returned by **CMXAllocateColors**. (**CMXAllocateColors** sets *ncolors* to the number of color cells actually allocated to you.) The arrays *r*, *g*, and *b* contain *len* many integers in the range 0−65535. The arrays can be either colormaps you have written yourself or one of the standard colormaps provided by CMX11. (See Section 2.5.3.)

Copyright © 1993 Thinking Machines Corporation

To set an individual color cell, use **CMXStoreColor**:

From CM Fortran:

```
SUBROUTINE CMXStoreColor(display, cmap, index, r, g, b)

INTEGER display
INTEGER cmap
INTEGER index
INTEGER r, g, b
```

From C*:

```
void CMXStoreColor(Display *display, Colormap cmap,
                   int index, int r, int g, int b);
```

NOTE

CMXStoreColor is simply a wrapper for the X11 routine **XStoreColor**. C* programmers can just call the X11 routine.

Remember that *index* must include the *base* offset into the colormap.

2.5.3 Creating Colormap Arrays

CMX11 provides a set of four standard colormaps that you can transfer to your colormap arrays with a call to **CMXLoadColorArrays**:

From CM Fortran:

```
      SUBROUTINE CMXLoadColorArrays(name, len, r, g, b)

      CHARACTER*(*) name
      INTEGER len
      INTEGER r(len)
      INTEGER g(len)
      INTEGER b(len)
CMF$ LAYOUT r(:SERIAL),b(:SERIAL),g(:SERIAL)
```


Copyright © 1993 Thinking Machines Corporation

From C*:

```
void CMXLoadColorArrays(char *name, int len,
                        int r[], int g[], int b[]);
```

The colormap *name* is one of the four standard colormaps:

- **`Density`**
  A scale from dark blue to cyan to yellow to red. This is the default colormap.

- **`Grayscale`** or **`Greyscale`**
  A linear sequence of gray shades from black in the first colormap entry to white at the last.

- **`Random`**
  Randomizes the entire red, green, and blue colormaps.

- **`Rainbow`**
  A sinusoidal distribution of colors ranging smoothly from black at the first index to red to green to blue and back to red. The intensity is constant throughout, only the hue changes.

These names are not case-sensitive: capitalization in the colormap name is ignored.

2.6 Setting the Foreground and Background Colors

CMX11 provides convenience functions with which you can set the foreground and background color of a graphics context (for drawing) directly from your CM Fortran or C* program:

From CM Fortran:

```
SUBROUTINE CMXSetForeground(display, gc, foreground)

INTEGER display
INTEGER gc
INTEGER foreground
```


Copyright © 1993 Thinking Machines Corporation

```
SUBROUTINE CMXSetBackground(display, gc, background)

INTEGER display
INTEGER gc
INTEGER background
```

From C*:

```
void CMXSetForeground(Display *display, GC gc,
                      int foreground);

void CMXSetBackground(Display *display, GC gc,
                      int background);
```

NOTE

CMXSetForeground and **CMXSetBackground** are simply wrappers for the X11 routines **XSetForeground** and **XSetBackground**. C* programmers can just call the X11 routines.

The color values are indices into the current colormap of the display. The foreground color is used for drawing by those CMX11 drawing routines that do not specify a color array:

CMXDrawPoint	**CMXDrawArc**
CMXDrawLine	**CMXFillArc**
CMXDrawRectangle	**CMXDrawString**
CMXFillRectangle	

The background color is used by CMX11 drawing routines that use a background color, such as **CMXPutImage1**. It does *not* set the background color for the window. To set the background color for the window, use **XSetWindowBackground**.

Copyright © 1993 Thinking Machines Corporation

2.7 Double Buffering

If your application involves animation or other frequent updates to the image, you can use double buffering to avoid the flicker usually seen when one image is written over another. Double buffering uses two image buffers; one buffer is used to display the image while the next updated image is drawn into the other buffer. When the updated image is complete, the buffers are switched so that the buffer containing the updated image is displayed and the display buffer becomes the draw buffer. This switch is synchronized with the regular screen refresh so that the updated image appears to replace the original image instantaneously. By repeatedly alternating buffers in this way, you can display a sequence of images without any perceptible flicker between images.

Double buffering depends on special display hardware that may not be supported on the display you are using. The CMX11 routine **CMXHasMBuffers** checks to see whether your server supports double buffering:

From CM Fortran:

```
LOGICAL FUNCTION CMXHasMBuffers (display)
INTEGER display
```

From C*:

```
logical CMXHasMBuffers(Display *display);
```

This routine returns FALSE if the specified display does not support double buffering and TRUE if it does. If you write your program to use the multibuffer routines and the display does not support true double buffering, CMX11 redraws the images into the display window as if no double buffering had been attempted.

If the display does support double buffering, you must allocate the buffers by calling **CMXAllocMBuffers**:

From CM Fortran:

```
INTEGER FUNCTION CMXAllocMBuffers(widget)
INTEGER widget
```

From C*:

```
Multibuffer* CMXAllocMBuffers(Widget widget);
```


Version 1.6, October 1993
Copyright © 1993 Thinking Machines Corporation

CMXAllocMBuffers makes a double buffer for the display associated with the *widget*, clearing it to the color of the background of the window from where you got *widget*. Then it returns an identifier to the multibuffer. You can pass this multibuffer return value as the *buffers* argument to **CMXGetDrawMBuffer** to get the identity of the buffer to be drawn into:

From CM Fortran:

```
INTEGER CMXGetDrawMBuffer(buffers)
INTEGER buffers
```

From C*:

```
Multibuffer CMXGetDrawMBuffer(Multibuffer *buffers);
```

The return value from **CMXGetDrawMBuffer** can then be used in any drawing routine where a *drawable* is required.

To make the current draw buffer the display buffer (make it visible) and the current display buffer the draw buffer, call **CMXSwitchMBuffers**:

From CM Fortran:

```
INTEGER CMXSwitchMBuffers(widget, buffers)
INTEGER widget
INTEGER buffers
```

From C*:

```
void CMXSwitchMBuffers(Widget widget,
                       Multibuffer *buffers);
```

CMXSwitchMBuffers clears the buffer that is to become the draw buffer. Therefore, the entire image should be redrawn on each frame.

You can then repeat the call to **CMXGetDrawMBuffer** to get the new draw buffer.

For example, the basic loop to implement double buffering with **CMXDrawPoint** would be

```
DRAWBUFFER = CMXGetDrawMBuffer(buffers)
CMXDrawPoint(display, DRAWBUFFER, gc, x, y)
CMXSwitchMBuffers(widget, buffers)
```


Copyright © 1993 Thinking Machines Corporation

If you want to write directly to the displayed buffer, you can determine the display buffer by calling **CMXGetDisplayMBuffer**:

From CM Fortran:

```
INTEGER CMXGetDisplayMBuffer(buffers)
INTEGER buffers
```

From C*:

```
Multibuffer CMXGetDisplayMBuffer(Multibuffer
                                 *buffers);
```

Finally, when you have completed your animation, you should release the buffers by calling **CMXFreeMBuffers**:

From CM Fortran:

```
CALL CMXFreeMBuffers(widget, buffers)
INTEGER widget
INTEGER buffers
```

From C*:

```
void CMXFreeMBuffers(Widget widget,
                     Multibuffer *buffers);
```


Copyright © 1993 Thinking Machines Corporation