Chapter 4

Display and Color Modes

This chapter describes how colors are displayed, from a hardware and software perspective. It tells you how to use the color modes to control the way color is handled and how to set the current color for drawing.

- Section 4.1, “Color Display,” describes how monitors display color, then focuses on some details of the color display implementation.
- Section 4.2, “RGB Mode,” tells you how to use RGBmode, a color mode that lets you work with the red, green, and blue components of color.
- Section 4.3, “Gouraud Shading,”tells you how to use Gouraud shading, which allows you to draw smoothly shaded figures.
- Section 4.4, “Color Map Mode,” tells you how to use color maps.
- Section 4.5, “Onemap and Multimap Modes,” tells you how to use alternate mapping modes.
- Section 4.6, “Gamma Correction,” tells you how to alter the colors your monitor displays and how to correct for display variations among different monitors.

4.1 Color Display

If you have a standard monitor, it has three color guns that *sweep* out the entire screen area 60 to 76 times per second. During this sweep, each gun points directly at each of the pixels for a very short time. The color guns shoot out electrons that strike the screen and cause it to glow.

Each pixel on the screen is composed of three different *phosphors* that glow red, green, or blue. One color gun activates only the red phosphors, one only the green, and the third only the blue. As each color gun sweeps across the pixels, the number of electrons shot out (the intensity) is modified on a pixel-by-pixel basis.

If no electrons are fired at a pixel, its phosphors do not glow at all, and it appears black. For example, consider the red color gun. If the gun is turned on to its highest intensity, the phosphor glows bright red. At intermediate intensities, the colors vary between the two—from black to bright red. The same is true for the other guns, except that the colors vary from black to bright green, or from black to bright blue.

The color your eye perceives at a pixel is the combination of all three colors. Different combinations of intensity settings of the guns generate a wide variety of colors.

Each color gun can be set to 256 different intensity levels, ranging from completely off to completely on. Setting 0 is completely off, and setting 255 is completely on. The intensities of the red, green, and blue guns at the pixel determine its color. This is expressed as an *RGB triple*: three numbers between 0 and 255 indicating the red, green, and blue intensity, in that order.

Black is represented by (0,0,0), bright red by (255,0,0), bright green by (0,255,0), and so on. A few other examples include: (255,255,0) = yellow, (0,255,255) = cyan, (255,0,255) = magenta, (255,255,255) = white. The colors represented by (0,0,0), (1,1,1), (2,2,2),..., (255,255,255) are different shades of gray ranging from black to white. Because each gun has 256 different settings, there are 256x256x256 = 16777216 different colors available.

Note: For a more detailed discussion of color, see J.D. Foley and V. Van Dam, *Computer Graphics: Principles and Practice*, Addison-Wesley, 1990. Also, the first example in Section 4.2 allows you to experiment with the correspondence between RGB triples and perceived colors.

4.1.1 Bitplanes

The framebuffer is organized as a set of *bitplanes*. Each bitplane contains exactly one bit of information for each pixel on the screen. The number of bitplanes varies from system to system.

Table 4-1 shows the bitplane configurations possible for different systems. C stands for color map mode; X means that the configuration is not supported.

Color Mode	IRIS Indigo	Personal IRIS/G			GTB GTXB VGXB XS24 Elan	GT/GTX VGX/VGXT SkyWriter	Reality Engine
RGB	8	8	12	24	24	32	32 / 48
RGB doublebuffer	4	X	X	12	24	32	32 / 48
Colormap	8	8	12	12	12	12	12
Colormap doublebuffer	4	4	8	12	12	12	12
Colormap multimap	8	8	8	8	8	8	8
C-multimap-double	4	4	8	8	8	8	8

Table 4-1 Bitplane Configurations of Silicon Graphics Workstations

These bits store color information, depth information, described in Chapter 8, overlays and underlays, described in Chapter 11, and, on systems with alpha planes, an alpha channel, described in Chapter 15.

The number of bits per pixel also depends on the color mode, RGB, colormap, or multimap, described later in this chapter; and on the drawing mode, normal, underlay, overlay, described in Chapter 11.

RealityEngine systems feature a flexible framebuffer configuration that gives you different pixel depths in different situations. See Chapter 15 for more information about the RealityEngine framebuffers.

Use `getgdesc()` with the `GD_BITS` group of inquiry parameters to determine exact bit availability. See the *getgdesc*(3G) man page for details on the inquiry parameters.

The creation of graphics includes two basic steps. First, the drawing subroutines write data into the bitplanes. Second, the display hardware interprets that data as colors on the screen. GL subroutines control both the patterns of zeros and ones that are written into the bitplanes of each pixel and the interpretation of those patterns as colors on the screen.

4.1.2 Dithering

The Personal IRIS and IRIS Indigo have less bitplanes per R, G, and B component than other IRIS systems, yet can still display a wide range of colors.

These systems use *dithering* to expand the range of displayed colors. Dithering is a pixel operation that attempts to convey the color of an image as accurately as possible when the framebuffer stores fewer bits of a color than the system calculates. Dithering creates a dot pattern of different colors from the colors available that looks like an accurately shaded image when viewed from a distance.

Dithering works on all drawing primitives: points, lines, text, polygons, and pixel write and copy operations. Dithering is enabled by default and works in both RGB mode and color map mode. In RGB mode, dithering is performed independently for each red, green, blue (and alpha) component.

Turn dithering on, which is the default on systems that support dithering, by calling `dither(DT_ON)`. Call `dither(DT_OFF)` to turn dithering off.

It is probably best to turn dithering off when drawing pre-dithered images, or when drawing single-width RGB lines on an eight or four bit framebuffer.

You can enable dithering on the Personal IRIS only when drawing in one of these modes: `shademodel(GOURAUD)` or `depthcue(TRUE)`.

Note: Some systems do not support dithering. Use `getgdesc(GD_DITHER)` to determine whether or not dithering is supported. If `getgdesc(GD_CIFRACT)` is FALSE, then dithering is only supported in RGB mode; it is not supported in color map mode.

4.2 RGB Mode

RGB mode specifies colors in terms of their components: red, green, blue, and alpha. The number of bits per component depends upon the system type.

Note: Regardless of the actual number of bits-per-pixel stored in the framebuffer, the GL allows you to treat the hardware as though it stores 24 bits per pixel. It does this by storing the most significant bits (MSBs) of the color components presented to it. This R, G, and B data can be treated as 8-bit values (0-255) by GL programs.

Some precision is lost in the RGB value by not having the other bits, but 2 bits of red data is still red, just not as precise as a shade of red described by 8 bits.

RealityEngine systems may compute and store 12 bits per color component and dither them into 10 bits when displaying.

The sample programs you have seen in the previous chapters take only a single color argument, for example, BLACK or GREEN. This is the default, color map mode, which is discussed later in this chapter.

Set the color mode to RGB mode with `RGBmode()` before you call any subroutines that require RGBmode. Call `gconfig()` to enable the mode change before you call any subroutines that use RGB data. You only need to call `RGBmode()` once, because the mode remains the same until you change it.

The following sample program, draws 64 squares in RGB mode and shows some of the colors possible. Each square is labeled with its RGB triple below it.

```
#include <stdio.h>
#include <gl/gl.h>

#define     R       0
#define     G       1
#define     B       2
#define     RGB     3
#define     XSPACING 120            /* strwidth(" 255 255 255 ") */
#define     SIZE    50
#define     SEP     30
#define     YSPACING (SIZE + SEP)

short whitevec[RGB] = {255, 255, 255};
short blackvec[RGB] = {0, 0, 0};
main()
```

```
{
    int i, j, k;
    Icoord majorx, majory;
    long xoff, yoff;
    char str[20];
    short rgbvec[RGB];
    if (getgdesc(GD_BITS_NORM_SNG_RED) == 0) {
    fprintf(stderr, "Single buffered RGB not available on this machine\n");
    return 1;
    }
    prefsize(8*XSPACING + SEP, 8*YSPACING + SEP);
    winopen("rgbdemo");
    RGBmode();
    gconfig();
    c3s(blackvec);
    clear();
    yoff = getheight();
    for (i = 0; i < 4; i++) {
        rgbvec[R] = i*255/3;
        if (i < 2)
            majory = SEP;
        else
            majory = SEP + YSPACING*4;
        if (i == 0 || i == 2)
            majorx = SEP;
        else
            majorx = SEP + XSPACING*4;
        for (j = 0; j < 4; j++) {
            rgbvec[G] = j*255/3;
            for (k = 0; k < 4; k++) {
                rgbvec[B] = k*255/3;
                c3s(rgbvec);
                sboxfi(majorx + XSPACING*j,  majory + YSPACING*k,
                majorx + XSPACING*j + SIZE,  majory + YSPACING*k + SIZE);
                c3s(whitevec);
                sprintf(str, "%d %d %d", rgbvec[R], rgbvec[G], rgbvec[B]);
                xoff = (strwidth(str) - SIZE)/2;
                cmov2i(majorx + XSPACING*j - xoff,
                        majory + YSPACING*k - yoff);
                charstr(str);
            }
        }
    }
    sleep(10);
    gexit();
    return 0;
}
```

The `c3s()` subroutine sets the color to the RGB triple specified by *blackvec*, which is initialized to contain zeros for the red, green, and blue components. The first location is the red component, the second is the green, and the third is the blue component. In the four-component case, the fourth component is called the *alpha* value, which is described in Chapter 15. For this example, set the fourth component to 255 (1.0 for floating point data).

Each color is then built up to contain the requisite red, green, and blue components, then filled rectangles are drawn in that color.

The *sprintf* command is a standard C library routine that is used here to create an ASCII string representation of the three values of red, green, and blue.

4.2.1 Setting and Getting the Current Color in RGB Mode

Like the vertex subroutines described in Chapter 2, the `c()` subroutines that set the current color in RGB mode have different forms depending on their data type.

Table 4-2 lists the subroutines that set the RGB color.

Argument Type	RGB	RGBA
16-bit integer	`c3s()`	`c4s()`
32-bit integer	`c3i()`	`c4i()`
32-bit floating point	`c3f()`	`c4f()`

Table 4-2 The Color (c) Family of Subroutines

The floating point range from 0.0 to 1.0 is mapped linearly to the range from 0 to 255, with values larger than 1.0 mapped to 255.

Note: On RealityEngine systems, colors are mapped to the range 0 to 4095.

You can also specify RGB component with `cpack()`. `cpack()` takes a single 32-bit argument that contains the packed 8-bit values of the red, green, blue, and alpha components. The red component is the low order 8 bits, green is next, then blue, and alpha is the high order bits. For example, `cpack(0x01020304)` sets the red, green, blue, and alpha components to 4, 3, 2, and 1, respectively.

Use `gRGBcolor()` to get the current RGB values while in RGB mode.

The old-style subroutine `RGBcolor()` was used to set the RGB color in early versions of the software. `RGBcolor()` is supported for compatibility only; it is recommended that you use the `c3i()`, `c3s()`, `c3f()`, or `cpack()` subroutines.

This sample program, *RGBcolor.c*, demonstrates how to use `RGBcolor()`:

```
#include <stdio.h>
#include <gl/gl.h>

main()
{
    if (getgdesc(GD_BITS_NORM_SNG_RED) == 0) {
    fprintf(stderr, "Single buffered RGB not available on this machine\n");
    return 1;
    }
    prefsize(400, 400);
    winopen("RGBcolor");
    RGBmode();
    gconfig();
    RGBcolor(0, 100, 200);
    clear();
    sleep(10);
    gexit();
    return 0;
}
```

4.3 Gouraud Shading

In the examples presented thus far, all lines and polygons are drawn in a uniform color—every pixel in the line or polygon has the same color. Uniformly colored polygons are called *flat-shaded* polygons. If the geometry you are drawing has only flat (polygonal) faces, this might be the way it should look; however, in many cases, polygonal faces are used to approximate a smooth curved surface. If the true surface is curved, colors tend to vary in a continuous way across the surface. A flat-shaded polygonal approximation to the surface has a tiled look.

One way to improve the appearance of an approximated polygonal surface is to use a large number of smaller polygons in the approximation. An easier way is to shade or vary the color across the polygons. This is called *Gouraud shading*.

It is possible to calculate the correct color for the true surface at each vertex of the approximating polygon, and the graphics hardware shades the polygon based on those values. (You can also shade polygons using lighting models, positions and colors of lights, and surface properties, as described in Chapter 9.) Lighting models calculate the colors only at polygon vertices, and the resulting shading is the same whether you or the lighting model hardware calculate the vertex colors.

Note: The graphics hardware on every IRIS model, except the G series and Personal IRIS, performs rapid Gouraud shading of polygons. On these machines, you can use the `shademodel()` subroutine to improve performance when Gouraud shading is not required. Flat shading provides the fastest possible shading on these systems. Gouraud shading is the default, so you must use `shademodel(FLAT)` to disable Gouraud shading of polygons. Calling `shademodel(GOURAUD)` restores Gouraud shading. Use `getsm()` to return the current value of shademodel.

For Gouraud-shaded polygons, the system *linearly interpolates*—that is, it averages the color values for each edge, assuming a constant variation between the RGB colors at the two vertices that delimit that edge. Next, it interpolates these colors from edge to edge across the interior of the polygon. The result is a smooth color variation across the entire polygon.

For Gouraud-shaded lines, the system uses the same process, except that only the first step—interpolating the color between the endpoints—is required.

The interpolation is linear in all three components.

Figure 4-1 shows the result of Gouraud shading a triangle whose vertices have colors (0,20,100), (75,60,50), and (0,0,0).

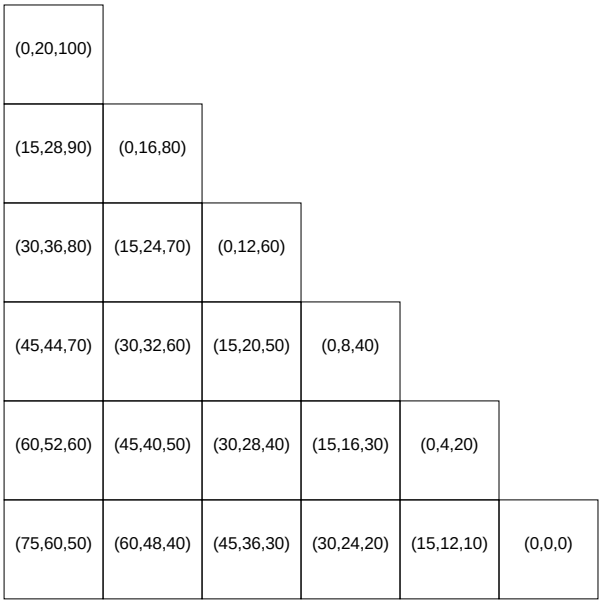

Figure 4-1 Gouraud Shaded Triangle

Look at the left edge of the triangle, where the color goes from (0,20,100) at one end to (75,60,50) at the other. The six pixels are colored (0,20,100), (15,28,90), (30,36,80), (45,44,70), (60,52,60), and (75,60,50). Each of the color components changes smoothly from one pixel to the next. The red component increases by 15 each time, the green component increases by 8, and the blue component decreases by 10 for each pixel. In this case, the pixel color differences happen to work out nicely to whole numbers. Usually this is not the case, but the approximation is done as accurately as possible.

After the colors of the pixels on the edges of the polygon are determined, the same process is used to interpolate the colors of the pixels on the interior.

The following sample program, *shadedtri.c*, draws a large Gouraud shaded triangle whose vertices are bright red, bright green, and bright blue.

```
#include <stdio.h>
#include <gl/gl.h>
#include <gl/device.h>

float blackvect[3] = {0.0, 0.0, 0.0};
float redvect[3] = {1.0, 0.0, 0.0};
float greenvect[3] = {0.0, 1.0, 0.0};
float bluevect[3] = {0.0, 0.0, 1.0};

long triangle[3][2] = {
   { 20, 20},
   { 20, 380},
   {380, 20}
};

main()
{
   if (getgdesc(GD_BITS_NORM_SNG_RED) == 0) {
   fprintf(stderr, "Single buffered RGB not available\n");
   return 1;
   }
   prefsize(400, 400);
   winopen("shadedtri");
   RGBmode();
   gconfig();

   c3f(blackvect);
   clear();
   bgnpolygon();
      c3f(redvect);
      v2i(triangle[0]);
      c3f(greenvect);
      v2i(triangle[1]);
      c3f(bluevect);
      v2i(triangle[2]);
   endpolygon();
   sleep(10);
   gexit();
   return 0;
}
```

When you run this program, notice how the colors change smoothly along each edge, and across the interior of the triangle. You can modify the colors at

the triangle's vertices to see how they affect the picture. Typically, polygons do not have wildly different colors at their vertices as the example does.

To shade a polygon with Gouraud shading, you set the color before each vertex. To shade a polyline, do the same thing—set the color before each vertex between `bgnline()` and `endline()`.

Note: You cannot change the color of a polyline on IRIS G series systems.

The next sample program, *cylinder.c*, implements a simple lighting model in user code and uses it to draw a cylinder. You normally would not program your own lighting calculations, you would take advantage of the built-in lighting facility of the IRIS GL.

```
#include <stdio.h>
#include <math.h>
#include <gl/gl.h>

#define X       0
#define Y       1
#define Z       2
#define XYZ     3
#define R       0
#define G       1
#define B       2
#define RGB     3

#define RADIUS .9
#define MAX(x,y) (((x) > (y)) ? (x) : (y))

float blackvec[RGB] = {0.0, 0.0, 0.0};
float lightpos[XYZ] = {3.0, 0.0, 1.2};
float dot(), dist2();

main()
{
    int i, nperside;
    float p[4][XYZ];
    float n[4][XYZ];
    float c[4][RGB];
    float x, dx;
    float theta, dtheta;
    float comp;

    if (getgdesc(GD_BITS_NORM_SNG_RED) == 0) {
        fprintf(stderr, "Single buffered RGB not available on this machine\n");
```

```
        return 1;
    }
    prefsize(400, 400);
    winopen("cylinder");
    RGBmode();
    gconfig();
    ortho2(-1.5, 1.5, -1.5, 1.5);
    c3f(blackvec);
    clear();

    for (nperside = 1; nperside < 10; nperside++) {
        dx = 3.0 / nperside;
        dtheta = M_PI / nperside;
        for (x = -1.5; x < 1.5; x = x + dx) {
            for (theta = 0.0; theta < M_PI; theta += dtheta) {
                p[0][X] = p[1][X] = x;
                p[0][Y] = p[3][Y] = RADIUS * fcos(theta);
                p[0][Z] = p[3][Z] = RADIUS * fsin(theta);
                p[2][X] = p[3][X] = x + dx;
                p[1][Y] = p[2][Y] = RADIUS * fcos(theta + dtheta);
                p[1][Z] = p[2][Z] = RADIUS * fsin(theta + dtheta);
                for (i = 0; i < 4; i++) {
                    n[i][X] = 0.0;
                    n[i][Y] = p[i][Y] / RADIUS;
                    n[i][Z] = p[i][Z] / RADIUS;
                }

                for (i = 0; i < 4; i++) {
                    comp = dot(lightpos, n[i])/(0.5 + dist2(lightpos, p[i]));
                    c[i][R] = c[i][G] = c[i][B] = MAX(comp, 0.0);
                }

                bgnpolygon();
                for (i = 0; i < 4; i++) {
                    c3f(c[i]);
                    v3f(p[i]);
                }
                endpolygon();
            }
        }
        sleep(5);
    }
    sleep(10);
    gexit();
    return 0;
}
```

```
/* dot: find the dot product of two vectors */
float dot(v1, v2)
float v1[XYZ], v2[XYZ];
{
    return v1[0]*v2[0] + v1[1]*v2[1] + v1[2]*v2[2];
}


/* dist2: find the square of the distance between two points */
float dist2(v1, v2)
float v1[XYZ], v2[XYZ];
{
    return (v1[0] - v2[0])*(v1[0] - v2[0]) +
            (v1[1] - v2[1])*(v1[1] - v2[1]) +
            (v1[2] - v2[2])*(v1[2] - v2[2]);
}
```

Although this program looks complicated, it is simple for a program that does its own lighting calculations. The program approximates a half cylinder by a set of $n \times n$ rectangles whose vertices lie on the surface of the cylinder.

The equation for the cylinder is $y^2 + z^2 = R^2$, where y and z are parameterized by θ: $y = cos(\theta)$, $z = sin(\theta)$, $0 \leq \theta \leq \pi$, and $-1.5 \leq x \leq 1.5$. Because you fix your eye above the middle of the cylinder, there is no need to draw the bottom half. Given x and θ, you can define a point on the surface as: (x, Rcos(q), Rsin(q)). At that point, the normal vector is (0, cos(q), sin(q)).

Assume that the cylinder is in a completely black room whose walls reflect no light, and there is a single point light source at (3.0, 0.0, 1.2), near the right end and above the center of the cylinder. Your eye is directly above the center of the cylinder and is looking straight down on it. This position is set by `ortho2()` in the example.

The cylinder is uniformly white, so you see only the colors between white and black, that is, only shades of gray where the red, green, and blue components of the light are equal. This model assumes that the amount of light reaching your eyes from a point on the cylinder depends on the distance of the light source to the point on the cylinder, and on the angle it makes with the cylinder's surface. The larger the angle, the less light is reflected. If the angle is more than 90 degrees, assume the color is black. The angular dependence is given by the dot product of the light direction with the cylinder's normal vector, and that is attenuated by a factor of $1.0 / (.5 + dist^2)$. This is not a realistic model, but it serves for this example. See Chapter 9 to learn how to use the built-in lighting models.

As written, the program loops on *n*, approximating the cylinder half first with one polygon, then 4, 9, 16, 25,... 100 polygons. Each view is presented for five seconds before the next one is drawn. The first view is completely black, because the normal vectors are all perpendicular to the light vector, so each corner is colored black. As the number of approximating polygons increases, the representation gets better, and the last two or three images are similar.

The points *p0*, *p1*, *p2*, and *p3* are the vertices of each of the approximating rectangles, *n0...n3* are the corresponding normal vectors, and *c0... c4* are the colors calculated at the points.

You can modify the program above to use different light vector positions or different lighting models. You can also modify it to draw flat shaded polygons, perhaps based on the normal vector at the center, to compare with the Gouraud shaded version. To get comparable pictures, many more polygons would have to be drawn.

4.4 Color Map Mode

In RGB mode, the values in the bitplanes correspond exactly to the color to be presented. Another way to write and interpret the data in the bitplanes is by using *color map mode*, also called *color index mode*. Many applications are better suited to color map mode than to RGB mode, and many of the principles of color maps are used in the overlay, underlay, and pop-up drawing modes.

Color map mode provides a level of indirection between the values stored in the bitplanes and the RGB values displayed on the screen. This mode is useful on systems that do not have enough bitplanes for RGB mode, and for blinking and other applications where you want quick color map changes.

In color map mode, the zeros and ones stored in the standard bitplanes (up to 12 bits) are interpreted as a binary number and used as an index into a color map. Each entry in the color map consists of a full 8 bits each of red, green, and blue intensity. To figure out what color to present at a pixel on the screen, take the 12 bits out of the bitplanes, interpret them as a binary number between 0 and 4095, and look up the color map values for red, green, and blue for that number. That red, green, and blue triple is the pixel color.

By default, the system is in color map mode, and the lowest eight values in the color map are loaded as shown in Table 4-3:

Location	Red	Green	Blue	Color
0	0	0	0	BLACK
1	255	0	0	RED
2	0	255	0	GREEN
3	255	255	0	YELLOW
4	0	0	255	BLUE
5	255	0	255	MAGENTA
6	0	255	255	CYAN
7	255	255	255	WHITE

Table 4-3 Default Color Map

For example, the call

color(GREEN);

actually sets the current color to 2, so every drawing subroutine (lines, points, polygons, or characters) puts the value 2 in the affected bitplanes. Because the display is in color map mode, pixel values of 2 are looked up in the color map, and the RGB triple (0,255,0) = bright green is presented.

These color map entries or any other color map entries can be changed. Because there is only one color map used by all GL polygons, whenever you modify the map, it is modified for all the other GL applications running in different windows. However, applications running in RGB mode, and non-GL applications such as X applications, are not affected.

To enter color map mode, call `cmode()` followed by `gconfig()`. The system is in color map mode by default, so you need only call `cmode()` if the system is currently in RGB mode.

To change color map entries, use `mapcolor()`. `mapcolor()` takes four arguments: an index into the color map, and the red, green, and blue components to load into the map for that index. The index is between 0 and the

system's limit, and the red, green, and blue components are integers between 0 and 255.

The calling sequence is:

```
mapcolor(index, red, green, blue);
```

Multimap mode has only 256 map entries. See Section 4.5, "Onemap and Multimap Modes."

4.4.1 Setting the Current Color in Color Map Mode

The `color()` and `colorf()` statements set the color index in the current draw mode of the active framebuffer. The framebuffer must be in color map mode for `color()` to work. Because most drawing commands copy the current color index into the color bitplanes of the current framebuffer, the system retains the value of `color()` for each framebuffer when you exit and reenter a particular draw mode.

`color()` takes a single argument, *c*, which represents a color index. *c* can have a value between 1 and 2^n-1, where *n* is the number of bitplanes available in the current draw mode. Color indices larger than 2^n-1 are clamped to 2^n-1; color indices smaller than 0 yield undefined results.

`colorf()` is identical to color, except that it uses a floating point value. Before the value is written into memory, however, it is rounded to the nearest integer value. The results of `color()` and `colorf()` are indistinguishable when using `shademodel(FLAT)`.

If you are using Gouraud shading, systems that iterate color indices with fractional precision yield more precise shading results with `colorf()` than with `color()`. Not all systems iterate with fractional precision. To determine whether your system supports fractional iteration, use `getgdesc(GD_CIFRACT)`.

Note: Do not call `color()` or `colorf()` when the framebuffer is in RGB mode.

This sample program, *pink-n-gray.c*, draws a gray rectangle around a pink circle. Here GRAY and PINK are indices into the color map. They are chosen to be larger than 63 so as not to conflict with the color map entries used by the window system. Note that the predefined color BLACK is used to clear the window.

```
#include <gl/gl.h>

#define GRAY 64
#define PINK 65

main()
{
 prefsize(400, 400);
 winopen("pink-n-gray");
 mapcolor(GRAY, 150, 150, 150);
 mapcolor(PINK, 255, 80, 80);
 color(BLACK);
 clear();
 color(GRAY);
 sboxi(150, 150, 250, 250);
 color(PINK);
 circi(200, 200, 50);
 sleep(10);
 gexit();
 return 0;
}
```

4.4.2 Getting Color Information in Color Map Mode

The GL provides two subroutines to get color information in color map mode: `getcolor()`, and `getmcolor()`. In most cases, `getcolor()` simply returns the most recently set color o; however, when using the fast update facility of the automatic lighting models, the system can change the current color as a result of lighting calculations (see the `lmcolor()` command, described in Chapter 9).

`getcolor()` returns the current color for the current drawing mode. It returns the index into the color map set by color. The result of `getcolor()` is undefined in RGB mode. Use `getcolor()` only in color map mode.

`getmcolor()` returns the setting of the color map for a given index. `getmcolor()` returns the red, green, and blue components of a color map entry for a given index into the color map.

4.4.3 Gouraud Shading in Color Map Mode

Gouraud shading works in color map mode, but it is more difficult to use than in RGB mode. The colors at the vertices of lines or polygons are interpolated to the interior points, but only the color map index is interpolated, not the red, green, and blue components. Thus, a shaded 6-pixel line whose endpoints are colored 1 (red) and 6 (cyan) has its six pixels colored 1, 2, 3, 4, 5, 6 (red, green, yellow, blue, magenta, cyan, respectively), assuming that the default color map is used. To shade in color map mode, you must first load a portion of the color map with a color ramp to use as gradient between the end colors.

The following sample program, *rampshade.c,* draws a Gouraud shaded polygon in color map mode.

```
#include <gl/gl.h>

#define RAMPBASE 64     /* avoid the first 64 colors */
#define RAMPSIZE 128
#define RAMPSTEP (255 / (RAMPSIZE-1))
#define RAMPCOLOR(fract) \
    ((Colorindex)((fract)*(RAMPSIZE-1) + 0.5) + RAMPBASE)

float v[4][3] = {
    {50.0, 50.0, 0.0},
    {200.0, 50.0, 0.0},
    {250.0, 250.0, 0.0},
    {50.0, 200.0, 0.0}
};

main()
{
 int i;

    prefsize(400, 400);
    winopen("rampshade");
    color(BLACK);
    clear();
/* create a red ramp */
    for (i = 0; i < RAMPSIZE; i++)
        mapcolor(i + RAMPBASE, i * RAMPSTEP, 0, 0);
    bgnpolygon();
        color(RAMPCOLOR(0.0));
        v3f(v[0]);
        color(RAMPCOLOR(0.25));
        v3f(v[1]);
```

```
        color(RAMPCOLOR(1.0));
        v3f(v[2]);
        color(RAMPCOLOR(0.5));
        v3f(v[3]);
    endpolygon();
    sleep(10);
    gexit();
    return 0;
}
```

The next sample program, *dithershade.c*, is the same as the previous program, but it uses dithering to interpolate the color shades between BLACK and RED.

```
#include <stdio.h>#include <gl/gl.h>

float v[4][3] = {
    {50.0, 50.0, 0.0},
    {200.0, 50.0, 0.0},
    {250.0, 250.0, 0.0},
    {50.0, 200.0, 0.0}
};

main()
{
    if (getgdesc(GD_CIFRACT) == 0) {
    fprintf(stderr, "Dithering not available "
           "on this machine\n");
    return 1;
    }
    prefsize(400, 400);
    winopen("dithershade");
    color(BLACK);
    clear();
    bgnpolygon();
        colorf(BLACK + 0.0);
        v3f(v[0]);
        colorf(BLACK + 0.25);
        v3f(v[1]);
        colorf(BLACK + 1.0);       /* = RED */
        v3f(v[2]);
        colorf(BLACK + 0.5);
        v3f(v[3]);
    endpolygon();
    sleep(10);
    gexit();
    return 0;
}
```

4.4.4 Blinking

Blinking is an advanced topic that can be skipped on the first reading.

The `blink()` subroutine changes a color map entry at a specified rate. It specifies a blink rate *(rate)*, a color map index *(i)*, and *red*, *green*, and *blue* values. The *rate* parameter indicates the number of *vertical retraces* (one sweep of the screen) at which the system updates the color located at *i* in the current color map. For every *rate* retrace, the color map entry *i* is remapped so that it alternates between the new *red*, *green*, *blue* value and the original values.

The following sample program, *blinker.c*, demonstrates blinking.

```
#include <gl/gl.h>

#define MAXBLINKS       20      /* maximum number of blinking entries */
#define FIRSTBLINKCI    64      /* avoid the first 64 colors */

main()
{
    int i;

    prefsize(400, 400);
    winopen("blinker");
    ortho2(-0.5, 20.0*MAXBLINKS + 9.5, -0.5, 500.5);
    color(BLACK);
    clear();
    for (i = MAXBLINKS - 1; i >= 0 ; i--) {
        mapcolor(i + FIRSTBLINKCI, 255, 255, 255);
        color(i + FIRSTBLINKCI);
        sboxfi(i*20 + 10, 10, i*20 + 20, 490);
        blink(i + 1, i + FIRSTBLINKCI, 255, 0, 0);
    }
    sleep(10);
    blink(-1, 0, 0, 0, 0);/* stop all blinking */
    gexit();
    return 0;
}
```

You can set up to 20 colors blinking simultaneously, each at a different rate; the 20-color limit is for each system, not for each window. You can change the blink rate by calling `blink()` a second time with the same *i* but a different *rate*.

To terminate blinking and restore the original color, call `blink()` with *rate* = 0 where *i* specifies a blinking color map entry. To terminate blinking of all colors,

call `blink()` with *rate* = -1. When you set *rate* to -1, the other parameters are ignored.

Note: Program termination does not stop the color map blinking; you must explicitly terminate blinking when you exit the program.

4.5 Onemap and Multimap Modes

Onemap and multimap modes are an advanced topic that you can skip on the first reading.

The default mode is `onemap()`; it organizes the color map as a single map with 4096 entries (256 on some systems). When you are in color map mode and in the default onemap mode, the value of the color bitplanes is used as an index into the color map to determine the color displayed on the screen.

An alternative mode, multimap mode, uses only 8 bits from the bitplanes (using 256 entries in the color map), but allows up to 16 completely independent 256-entry color maps.

`multimap()` organizes the color map as 16 small maps, each with a maximum of 256 RGB entries. Multimap mode is useful on systems with only 8 bitplanes (or 16 in double buffer mode; see Chapter 6), but it can be used on other systems for techniques such as color map animation.

In multimap mode, `setmap()` makes one of the 16 small color maps current. All display is done using the current small map, and `mapcolor()` affects that map.

You must call `gconfig()` to activate the `onemap()` or `multimap()` settings.

Use `getcmmode()` to return the current color map mode. FALSE indicates multimap mode; TRUE indicates onemap mode.

Use `getmap()` to return the number (from 0 to 15) of the current color map. In onemap mode, `getmap()` always returns 0.

Use `cyclemap()` to cycle through color maps at a specified rate. It defines a duration (in vertical retraces), the current map, and the next map that follows when the duration lapses.

For example, the following routines cycle between two maps in multimap mode, leaving map 1 on for ten vertical retraces and map 3 on for five retraces.

```
void cyclemap(duration, map, nextmap)
short duration, map, nextmap;
multimap();
gconfig();
cyclemap(10, 1, 3);
cyclemap(5, 3, 1);
```

Note: When you kill a window or attach to a new one, the maps stop cycling.

4.6 Gamma Correction

Gamma correction is an advanced topic that you can skip on the first reading.

The light output of any video display is controlled by the input voltage to the monitor. The relationship between input voltage and the brightness of the display is exponential rather than linear. For instance, assume that 100 percent of a monitor's input voltage produces 100 percent brightness. If you reduce the voltage to 50 percent of its initial value, the monitor displays only 19 percent of its initial brightness.

To achieve a linear response from the monitor, the system must vary the input voltage by an exponent. The exponent is called the monitor's *gamma*. Linear response is achieved on standard IRIS-4D monitors with a gamma of 2.4. The system uses a hardware look-up table to compensate for non-linear response.

Use `gammaramp()` to provide gamma correction, to equalize monitors with different color characteristics, or to modify the color warmth of the monitor. `gammaramp()` supplies another level of indirection for all color map and RGB values. Usually, the gamma ramp map is loaded with gamma corrections, but it can be loaded with any values. The default setting assigns a gamma exponent of 1.7.

`gammaramp()` affects the entire screen and all running processes. It affects only the display of color, not the values that are written in the bitplanes. The gamma correction stays in effect until another call to `gammaramp()` is made, or until the graphics hardware is reset.

Note: On IRIS-4D / G systems, the gamma ramp is stored in the top 256 entries of the color map.

The following sample program, *setgamma.c*, takes a floating point gamma value, calculates a standard gamma ramp, and installs it using `gammaramp()`.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <gl/gl.h>

main(argc, argv)
int argc;
char *argv[];
{
    int i;
    double val;
    short tab[256];

    if (argc != 2) {
        fprintf(stderr, "Usage: setgamma <value>\n");
        return 0;
}
    val = atof(argv[1]);

    noport();
    winopen("setgamma");
    for (i = 0; i < 256; i++)
        tab[i] = 255.0 * pow(i / 255.0, 1.0 / val) + 0.5;
    gammaramp(tab, tab, tab);
    gexit();
    return 0;
}
```

This program sets the same gamma ramp for red, green, and blue, although a more general mapping is possible. In addition, this program uses the `noport()` hint to the window manager in the same way that `prefsize()` does. It tells the graphics that no physical screen space is required, but that the graphics hardware will be accessed.

As a final example of the gamma ramp, this *stepgamma.c* program sets the gamma ramp to a set of discrete values. For example, if the number is 3, the highest third of the ramp is mapped to full intensity; the middle third to middle intensity, and the lowest third to lowest intensity. It basically provides simultaneous thresholding in red, green, and blue. If a picture is drawn

entirely with shades of gray, this loading of the gamma ramp displays the picture as if it were drawn with a small set of discrete gray values.

```
#include <stdio.h>
#include <stdlib.h>
#include <gl/gl.h>

main(argc, argv)
int argc;
char *argv[];
{
    int i, nsteps;
    short val;
    short ramp[256];

    if (argc != 2) {
        fprintf(stderr, "Usage: stepgamma <numsteps>\n");
        return 1;
    }
        nsteps = atoi(argv[1]);
        if (nsteps < 2) {
            fprintf(stderr, "stepgamma: <numsteps> cannot be < 2\n");
        return 1;
        }

    noport();
    winopen("stepgamma");
    for (i = 0; i < 256; i++) {
        val = ((nsteps * i) / 256) * 255 / (nsteps - 1);
        if (val > 255)
            val = 255;
            ramp[i] = val;
        }
    gammaramp(ramp, ramp, ramp);
    gexit();
    return 0;
}
```

It is interesting to look at some shaded RGB polygons with a small number of steps.