Graphics Library Programming Guide

IRIS-4D Series

SiliconGraphics
Computer Systems

Document number: 007-1210-020

Graphics Library Programming Guide

Document Version 2.0

Document Number 007-1210-020

5/90

Technical Publications:
Scott Fisher
Melissa Heinrich

Engineering:
Kurt Akeley
Rolf van Widenfelt
George Kong
Herb Kuta
Dave Ratcliffe
Linda Roy
Gary Tarolli
Vince Uttley

Graphics Library Programming Guide
Document Version 2.0
Document Number 007-1210-020

Silicon Graphics, Inc.
Mountain View, California

Contents

List of Figures

List of Tables

Introduction

The Graphics Library™ (GL) is a set of graphics and utility routines that provide high- and low-level support for graphics. This book is primarily organized as a reference guide to the Graphics Library. Thus, similar routines are grouped together, despite their complexity. Because you also need to use this book as a tutorial, all complex topics are so indicated, and can be skipped on the first reading. In addition, examples in later sections do not use the advanced material unless they also are marked as advanced.

The goal of the first chapter is to give you enough information to write simple graphics programs. By the end of the chapter, you will be able to open a graphics window and draw a few simple figures of different colors. The first chapter does not attempt to present all the details of the referenced subroutines.

This book assumes that you are familiar with Silicon Graphics, Inc.'s, operating system, IRIX™, and can create and edit files. If you are not familiar with IRIX, see *Getting Started with the IRIS-4D Workstation* , or see the *Personal IRIS Owner's Guide* if you have a Personal IRIS™. This book also assumes that you are sufficiently familiar with C to write a program that prints "hello, world" to the console.

Throughout this book, the word "IRIS" designates all models of the IRIS-4D™ Series of workstations, including the Personal IRIS. Where it is necessary to differentiate among the various workstations, the applicable product is called out by name.

This book does not assume that you have a knowledge of computer graphics. However, if you are already familiar with the basic concepts of computer graphics, you will find it easy to learn this particular implementation. To get an introduction to these concepts, read a standard computer graphics text, such as Newman and Sproull's *Principles of Interactive Computer Graphics*, or Foley and van Dam's *Fundamentals of Interactive Computer Graphics*.

How This Manual Is Organized

This manual is organized so that new Graphics Library programmers can read it from front to back, skipping over some of the more complex topics until later. You should try to run as many of the example programs as you can, so that you can see how the Graphics Library software functions.

Once you have tried the example programs or program segments, experiment with the Graphics Library functions to get the effects you want to use in your own programs. In many cases, you might be able to manipulate the examples in order to achieve the effects you want; in others, you will need to fit the Graphics Library functions into the structure of your own programs. The examples are general enough that you should be able to adapt them for your own needs without restricting your own programs.

Remember, in the interest of space, many of the examples are only fragments of a complete program. They presume that you have constructed the necessary program framework required for complete compilation and operation (such as the declaration of variables and the correct scoping of local and global variables for parameter passing).

Once you have worked through the book, you might find it a useful addition to the man pages in the *Graphics Library Reference Manual.* Though these are excellent sources for quickly accessing information about individual Graphics Library functions, this manual contains examples and other contextual information that can help you, particularly if you are unfamiliar with the overall concept (such as shading, curves, or window system interaction) represented by the Graphics Library function that concerns you.

Introductory Example Program

All the examples contained in this manual can be found on-line in the directory:

/usr/people/4Dgifts/examples/glpg/ch<nn>

where *nn* represents the chapter number.

The first example is the following program, called *green*:

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

main()
{
    prefsize(400, 400);
    winopen("green");
    color(GREEN);
    clear();
    sleep(10);
    gexit();
    return 0;
}
```

Compile the program using the compile line:

```
% cc green.c -lgl_s -lc_s -o green
```

`-lgl_s` links the loader with the shared Graphics Library; `-lc_s` links with the shared C library. These two options allow the same binary to run on all IRIS-4D Series systems. The last command line option, `-o`, defines that the output file uses the name `green`.

Run the program by typing:

```
% green
```

The window manager prompts you for the position of a window. Move the mouse cursor to the location where you want the window to appear and click any mouse button. A solid green window opens, remains visible for 10 seconds and then disappears.

Look at the program in detail. The first line,

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

includes all the standard definitions for graphics. It must be included in every graphics program. You must include *gl/gl.h* in *green.c* to get the definition of "GREEN" in the call to `color()`.

The subroutine:

```
prefsize(400,400);
```

tells the window system that when a window is opened, it should be 400 pixels on a side. It doesn't create a window—it just establishes the initial size of the next window to be opened.

The call to `winopen()` actually creates the window and causes the window manager to prompt you for the window's location. Calling `color()` with an argument of `GREEN` sets the drawing color for subsequent operations to the value of the constant `GREEN`, defined in *gl/gl.h*. The call to `clear()` fills the window with the current drawing color. You set the drawing color for other operations in the same manner: by calling `color()` with the color specification you wish to use. Color specifications and other drawing operations are described in detail elsewhere in the manual. The call to `gexit` closes the window and tells the system that the process is finished using graphics.

1. Controlling the Graphics Environment

This chapter introduces you to the graphics programming environment. It describes how to manipulate the system's global state attributes, that is, the software and hardware environment

To program on the system, you need to:

- initialize the Graphics Library
- terminate use of the Graphics Library
- query the availability of certain graphics capabilities
- change global state attributes

Initializing the Graphics Library means telling the system to activate the graphics software and hardware environment in which a program will run. Use `winopen` to initialize the Graphics Library.

The global state attributes are options that specify modes, line style, window specifications, and hardware requirements. Unless you specify otherwise, the global state attributes use their default values. (See Table 1–1 for the default values of the global state attributes.)

Attribute	Initial Value	Code**
acsize	0	G
afunction	0	AF_ALWAYS
backbuffer	FALSE	A
backface	FALSE	
blendfunction	BF_ONE, BF_ZERO	
character position	undefined	
clipplane	CP_OFF	
cmode	TRUE	A, G
color	0	A, V
concave	FALSE	
curveprecision	undefined	
depth range	*Zmin,Zmax****	
depthcue	FALSE	
doublebuffer	FALSE	G
drawmode	NORMALDRAW	A
feedback mode	off	
fogvertex	FG_OFF	
font	0*	A
frontbuffer	TRUE	A
frontface	FALSE	
full screen mode	off	
glcompat		
GLC_OLDPOLYGON	1	
GLC_ZRANGEMAP	1 (B and G models)	
	0 (other models)	
graphics position	undefined	
linesmooth	SML_OFF	
linestyle	0 (solid)	A
linewidth	1	A
lmcolor	LMC_COLOR	V
lmbind		
BACKMATERIAL	0	
LIGHT*n*	0	
LMODEL	0	
MATERIAL	0	
logicop	LO_SRC	
lsrepeat	1	A
mapcolor	no entries changed	

Table 1–1. Initial Values of Global State Attributes

Attribute	Initial Value	Code**
matrix		
ModelView	undefined	
Projection	undefined	
Single	`ortho2` matching window size	
Texture	undefined	
`mmode`	MSINGLE	
`multimap`	FALSE	G
name stack	empty	
`nmode`	NAUTO	
normal vector	undefined	V
`onemap`	TRUE	G
`overlay`	2	G
`patchbasis`	undefined	
`patchcurves`	undefined	
`patchprecision`	undefined	
pattern	0 (solid)	A
pick mode	off	
`picksize`	10x10	
`pixmode`	standard	
`pntsmooth`	SMP_OFF	
`polymode`	PYM_FILL	
`polysmooth`	PYSM_OFF	
`readsource`	SRC_AUTO	
`rectzoom`	1.0,1.0	
RGB color	all components 0 (when RGBmode is entered)	A,V
RGB shade range	undefined	
RGB writemask	0xFF (when RGB is entered)	A,V
`RGBmode`	FALSE	G A
`scrbox`	SB_RESET	
`scrmask`	set to size of window	
`scrsubdivide`	SS_OFF	
select mode	off	
shade range	0,7,*Zmin*,*Zmax****	
`shademodel`	GOURAUD	A
`singlebuffer`	TRUE	G
`stencil`	disabled	
`stensize`	0	G

Table 1–1 (continued). Initial Values of Global State Attributes

Attribute	Initial Value	Code**
swritemask	all stencil planes enabled	
tevbind	0 (off)	
texbind	0 (off)	
texgen	TG_OFF	
underlay	0	G
viewport	set to size of window	
writemask	all bitplanes enabled	A
zbuffer	FALSE	
zdraw	FALSE	A
zfunction	ZF_LEQUAL	
zsource	ZSRC_DEPTH	
zwritemask	all z-buffer planes enabled	

Table 1–1 (continued). Initial Values of Global State Attributes

Notes on table 1–1:

* Font 0 is a Helvetica-like font.

** Code: A—is pushed and popped on the attributes stack; G—takes effect when `gconfig` is called; V—can be changed between `bgn` and `end` calls (`bgnpoint`, `bgnline`, `bgnclosedline`, `bgnpolygon`, and `bgntmesh`)

***_Zmin_ and _Zmax_ are the minimum and maximum values that you can store in the z-buffer. These depend on the graphics hardware and are returned by `getgdesc(GD_ZMIN)` and `getgdesc`(GD_ZMAX).

1.1 Initializing a Program

The subroutines `ginit` and `gbegin` were used primarily in programs that did not run under the window manager. On current systems, these subroutines cause the window to occupy the entire screen. You might want to use `ginit` or `gbegin` to run programs written for older systems or, if you are a novice user, to run a program on the full screen to simplify the programming process.

winopen

`winopen` initializes the hardware, allocates memory for symbol tables and display list objects, and sets up default values for global state attributes (see Table 1–1).

`winopen` must be called before you call most Graphics Library (GL) routines. It performs the initialization functions listed in Table 1–1. `winopen` makes no change to any color map value, however.

Index	Name	RGB Value		
		Red	Green	Blue
0	BLACK	0	0	0
1	RED	255	0	0
2	GREEN	0	255	0
3	YELLOW	255	255	0
4	BLUE	0	0	255
5	MAGENTA	255	0	255
6	CYAN	0	255	255
7	WHITE	255	255	255
all others	unnamed		unchanged	

Table 1–2. Default Color Map Values

greset

`greset` returns the global state attributes to their initial values. You can call `greset` at any time. Table 1–1 lists the global state attributes and their default values.

`greset` initializes the first eight entries in the color map to the values shown in Table 1–2. `greset` also sets up a 2-D orthographic projection transformation that maps user-defined coordinates to the entire area of the window (see Chapter 7, "Coordinate Transformations").

```
void greset()
```

gexit

`gexit` performs housekeeping functions associated with the termination of graphics programming. It closes all windows and deletes display lists (see Chapter 16, "Graphical Objects") and other defined objects. It also turns off any blinking initiated by the program.

```
void gexit()
```

getplanes

`getplanes` returns the number of bitplanes that are available in the current hardware and software configuration and drawing mode. For more information, see Chapter 11, "Frame Buffers and Drawing Modes." Table 1–3 lists the possible combinations. C stands for color map mode; X means the configuration is not supported.

	System Model				
Mode	**Personal IRIS/G**			**GTB/GTXB/VGXB**	**GT/GTX/VGX**
RGB	8	12	24	24	32
RGB double	X	X	12	24	32
C	8	12	12	12	12
C double	4	8	12	12	12
C-multimap	8	8	8	8	8
C-multimap-double	4	8	8	8	8

Table 1–3. Bitplane Availability

```
long getplanes()
```

getgdesc

`getgdesc` allows you to inquire about characteristics of the graphics system and returns a description of part of the graphics system specified by its parameter, *inquiry*. `getgdesc` returns the numeric value of the requested characteristic, or -1 if it is invalid. You can call `getgdesc` at any time, including before the first `winopen`.

The values it returns depend only on the hardware configuration. They are are not affected by changes to software modes or software configuration.

```
long getgdesc(inquiry)
long inquiry;
```

Refer to the `getdesc` manual page for a complete list of the inquiries supported by the current software release.

gversion

`gversion` returns information about the current graphics hardware and the Graphics Library version. Its argument, *v*, expects a pointer to a location into which `gversion` copies a null-terminated string. Reserve at least 12 characters at this location.

`gversion` fills the buffer pointed to by *v* with a null-terminated string that specifies the graphics hardware type and the version number of the Graphics Library on the system in question, according to the information contained in Table 1–4. In the table, *m* and *n* represent the major and minor release numbers, respectively, of the IRIX software release to which the current Graphics Library belongs.

Graphics Type	String Returned
B or G	GL4D-*m.n*
GT or GTB	GL4DGT-*m.n*
GTX or GTXB	GL4DGTX-*m.n*
VGX	GL4DVGX-*m.n*
Personal IRIS	GL4DPI2-*m.n*
Personal IRIS with Turbo Graphics	GL4DPIT-*m.n*
Personal IRIS (early serial numbers)	GL4DPI-*m.n*

Table 1–4. System Types and Graphics Library Versions

You can call `gversion` before the first `winopen`.

Personal IRIS units with early serial numbers (see Table 1–4) do not support the complete Personal IRIS graphics functionality.

```
long gversion(v)
String v;
```

Caution: Using `gversion` makes programs machine-specific. In almost all cases, `getgdesc` is preferable to `gversion`.

glcompat

`glcompat` gives control over details of the compatibility between IRIS-4D Series models. `glcompat` controls two compatibility modes. The first, GLC_OLDPOLYGON, offers compatibility with old-style polygons (see section 2.8, "Old-Style Drawing"). The second, GLC_ZRANGEMAP, controls the state of z-range mapping mode.

The default value for GLC_OLDPOLYGON is TRUE—polygons are filled and outlined. This means that polygons drawn with the new GL routines have the same appearance as polygons drawn with the old GL routines. On all systems except the IRIS-4D/B/G, the new (point-sampled) polygon routines draw more quickly than the older outlined polygons. (See Chapter 2, "Drawing.") For maximum performance in cases where compatibility with older routines is not an issue, set GLC_OLDPOLYGON to FALSE. This mode applies to each window in which it is asserted; it is ignored on IRIS-4D/B/G systems.

The other `glcompat` mode, GLC_ZRANGEMAP, controls how z-buffer range is mapped on a given system. (See Section 8.2, "Controlling z values," for more information on the z depth of various IRIS-4D Series systems.) GLC_ZRANGEMAP can have either of two values: 0 or 1. When this mode is 0, the domain of the z-range arguments to `lsetdepth`, `lRGBrange`, and `lshaderange` depends on the graphics hardware. The minimum is the value returned by `getgdesc(GD_ZMIN)`, and the maximum is the value returned by `getgdesc(GD_ZMAX)`. When this mode is 1, these routines accept the range 0x0 to 0x7FFFFF, and the range is mapped to whatever range the graphics hardware supports. To maintain backwards compatibility, the default range of GLC_ZRANGEMAP is 1 on IRIS-4D/B/G systems, and 0 on all other systems. This mode applies to each process in which it is asserted.

```
void glcompat(mode, value)
long mode, value;
```

1.2 Saving Global State Attributes

`pushattributes` and `popattributes` manipulate a defined list of global state attributes. They operate on all attributes that are marked with an "A" in Table 1–1.

pushattributes

`pushattributes` saves a portion of the current global state on a stack that the system maintains.

```
void pushattributes()
```

popattributes

`popattributes` restores the most recently saved values of the global state attributes. If you attempt to pop an empty attributes stack, an error message appears.

```
void popattributes()
```

2. Drawing

This chapter describes the Graphics Library subroutines that draw many of the most basic and therefore most important geometric figures: points, lines, polygons, rectangles, circles, arcs, and meshes. Being geometric, all these figures can undergo three-dimensional transformations such as rotation and translation, and can then be viewed in perspective.

This chapter does not cover more complicated geometric figures such as curves and surfaces. Curves and surfaces are an advanced topic, and an entire chapter (Chapter 14) has been reserved for them. Neither does this chapter describe the Graphics Library subroutines for drawing things that are not geometric figures—for example, character strings and pixel data. For a discussion of the Graphics Library subroutines that handle text and fonts, see Chapter 3. For a discussion of the Graphics Library subroutines for use with pixel data, see Chapter 10.

In this chapter, the drawing subroutines are divided into three sections. The first section deals with the primitive subroutines on which all the others are based—points, lines, and polygons. These subroutines make up the high-performance library and are tuned to the hardware architecture. Use these subroutines when high performance is required. The second section describes several useful higher level subroutines that draw common objects such as rectangles, circles, and arcs. Finally, the third section covers the old-style graphics subroutines, which are included in the current library for compatibility. They are less efficient than the high-performance subroutines, but appear extensively in code written for earlier products.

All the drawing subroutines in this chapter are affected by various attributes that are discussed at length in later chapters. These attributes include color, texture pattern, line style (stippling pattern), and many other things. For example, if your program includes the line `color(RED);`, and you draw a line, the line will be red. If you draw a polygon, the polygon will be red, and so on, until you issue another `color` subroutine call. Similarly, if you set a line style, it remains in effect until you set another.

The examples in this chapter use other subroutines from the Graphics Library that are described in Chapter 1, "Controlling the Graphics Environment." Look over that chapter before beginning this one if you have not already done so.

2.1 High-Performance Drawing

High-performance drawing provides the fastest way to draw primitive graphical figures. Points, lines, and polygons are all described in terms of vertices (sets of coordinates that identify points in space). A point is described by a single vertex. A line segment is described by two vertices indicating its endpoints. A polygon is described by a set of three or more vertices indicating its corners.

To draw a graphical figure as efficiently as possible, use a series of vertex subroutines surrounded by a pair of begin and end subroutines, which mark the beginning and end of the figure. For example, the code to draw a set of five points A, B, C, D, and E takes the form:

```
<beginning of point vertices>
<vertex A>
<vertex B>
<vertex C>
<vertex D>
<vertex E>
<end of point vertices>
```

If instead, you wanted to draw a polygon whose corners are the same five points, the code would take the form:

```
<beginning of polygon vertices>
<vertex A>
<vertex B>
<vertex C>
<vertex D>
<vertex E>
<end of polygon vertices>
```

Only the `bgn/end` point sequence and the vertex subroutines outside of any `bgn/end` sequence leave the current graphics position defined. All other high-performance primitives leave the current graphics position undefined.

Between any bgn/end pair, you can use only these commands: c, color, cpack, lmbind, lmcolor, lmdef, n, RGBcolor, t, and v. lmbind and lmdef can be used only to respecify materials and their properties.

This program clears a window to white, and then draws a pair of red lines connecting its opposite corners.

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

long vert1[2] = {100, 100}; /* lower left corner */
long vert2[2] = {100, 500}; /* upper left corner */
long vert3[2] = {500, 500}; /* upper right corner */
long vert4[2] = {500, 100}; /* lower right corner */

main()
{
    prefposition(100, 500, 100, 500);
    winopen("crisscross");
    ortho2(99.5, 500.5, 99.5, 500.5);
    color(WHITE);
    clear();
    color(RED);
    bgnline();
        v2i(vert1);
        v2i(vert3);
    endline();
    bgnline();
        v2i(vert2);
        v2i(vert4);
    endline();
    sleep(3);
    gexit();
    exit(0);
}
```

In this example, you declare four `long` arrays *vert1*, *vert2*, *vert3*, and *vert4*, and assign values to all the elements of each array. `prefposition` defines the next window as a square covering pixels 100 through 500 in both the *x* and *y* directions. `winopen` then opens the window described by `prefposition` and assigns it the name "crisscross." `ortho2` sets up the default coordinate system so that a point with coordinates *(x, y)* maps exactly to the point on the screen that has the same coordinates. (The ortho2 command is discussed in Chapter 7.) `color` sets the window's color property to white and `clear()` clears the window to the current value of the window's color property, white.

The next four lines of code draw a line from (100, 100) to (500, 500)—the lower-left corner to the upper-right corner. `bgnline` tells the system to prepare to draw a line using the following vertices. `v2i` takes an array of coordinates as its argument and creates a vertex at those coordinates.

The first `v2i` subroutine call after `bgnline` creates the first endpoint of the line segment. The second `v2i` subroutine call after `bgnline` creates the endpoint of the line segment and the system draws a line. The `endline` subroutine call tells the system that it has all the vertices for the line. The next four lines draw a line from (100, 500) to (500, 100), the lower-right corner to the upper-left corner.

Finally, `sleep(3)` delays the program from exiting until three seconds pass; the picture remains on the screen for three seconds.

2.1.1 Polylines

If more than two points are listed between `bgnline` and `endline`, each point is connected to the next by a line. For example, the program below draws an outlined green square in the center of the window.

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

long vert1[2] = {200, 200};
long vert2[2] = {200, 400};
long vert3[2] = {400, 400};
long vert4[2] = {400, 200};

main()
{
    prefposition(100, 500, 100, 500);
    winopen("greensquare");
    ortho2(99.5, 500.5, 99.5, 500.5);
    color(WHITE);
    clear();
    color(GREEN);
    bgnline();
        v2i(vert1);
        v2i(vert2);
        v2i(vert3);
        v2i(vert4);
        v2i(vert1);
    endline();
    sleep(3);
    gexit();
    exit(0);
}
```

Notice that the first vertex, `v2i(vert1)`, is repeated to close the series of line segments.

A series of connected line segments is called a *polyline*. There are no restrictions on a polyline—the segments can cross each other, vertices can be reused, and if the vertices are defined in terms of three dimensions, you can place them anywhere within three-dimensional space. In a three-dimensional space, the vertices need not all lie in the same plane.

2.1.2 The Vertex Subroutine

The previous examples use only one form of the vertex subroutine—a two-dimensional version with 32-bit integer coordinates. The Graphics Library contains 12 forms of the vertex subroutine. The coordinates can be short integers (16 bits), long integers (32 bits), single-precision floating point values (32 bits), and double-precision floating point values (64 bits). For each of these types, there is a two-dimensional version, a three-dimensional version, and a version that expects vertices expressed in homogeneous coordinates (often referred to as a 4-D version because it takes four parameters to define a point in a three-dimensional space). If you don't know what homogeneous coordinates are, don't worry. Homogeneous coordinates are an advanced topic that you can read more about in Newman and Sproull's *Principles of Interactive Computer Graphics.*

The IRIS-4D Series converts all arguments to 32-bit floating point for hardware calculations; consequently, only long integers in the range of -2^{23} and $2^{23}-1$ are converted correctly. Treat long integers as 24-bit 2s-complement numbers sign-extended to 32 bits.

The vertex subroutines all have the same pattern, as illustrated in Table 2–1

Argument Type	2-D	3-D	4-D
16-bit integer	v2s	v3s	v4s
32-bit integer	v2i	v3i	v4i
32-bit floating point	v2f	v3f	v4f
64-bit floating point	v2d	v3d	v4d

Table 2–1. The Vertex Subroutine

All forms of the vertex subroutine begin with the letter "v." The second character is "2", "3", or "4" indicating the number of dimensions, and the final character is "s" for short integer, "i" for long integer, "f" for single-precision floating point, and "d" for double-precision floating point.

The following program illustrates the use of some of the different vertex subroutines. It draws exactly the same picture as the previous example does—it just uses different versions of the vertex subroutine.

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

short  vert1[3] = {200, 200, 0};
long   vert2[2] = {200, 400};
float  vert3[2] = {400.0, 400.0};
double vert4[3] = {400.0, 200.0, 0.0};

main()
{
    prefsize(400, 400);
    winopen("greensquare2");
    ortho2(99.5, 500.5, 99.5, 500.5);
    color(WHITE);
    clear();
    color(GREEN);
    bgnline();
        v3s(vert1);
        v2i(vert2);
        v2f(vert3);
        v3d(vert4);
        v3s(vert1);
    endline();
    sleep(10);
    gexit();
    return 0;
}
```

Although it is unlikely that you would write a program like the one above, it does illustrate two things:

- Within one geometric figure (in this case, a polyline), you can mix different kinds of vertices together. In a typical application, all the vertices would tend to have the same dimension and have the same form.

- In the Graphics Library, all geometric figures are three-dimensional and the hardware treats them as such. Two-dimensional versions of the vertex subroutines are actually shorthand for an equivalent three-dimensional subroutine with the *z* coordinate set to zero.

2.1.3 Closed Lines

In the last two examples, the program drew a closed polyline—a line segment connected the last point in the polyline to the first point in the polyline. Since this is a fairly common operation, there is a pair of high-performance subroutines to do it: `bgnclosedline` and `endclosedline`.

The following code draws a regular n-gon (n-sided polygon) centered at the origin:

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

main(argc, argv)
int argc;
char *argv[];
{
    int n, i;
    float vert[2];

    if (argc != 2) {
        fprintf(stderr, "Usage: n-gon <number of sides>\n");
        return 1;
    }
    n = atoi(argv[1]);

    prefsize(400, 400);
    winopen("n-gon");
    ortho2(-1.5, 1.5, -1.5, 1.5);
    color(WHITE);
    clear();
    color(RED);
    bgnclosedline();
    for (i = 0; i < n; i++) {
        vert[0] = fcos(i * 2.0 * M_PI / n);
        vert[1] = fsin(i * 2.0 * M_PI / n);
        v2f(vert);
    }
```

```
        endclosedline();
        sleep(10);
        gexit();
        return 0;
}
```

A couple of steps might need some explanation if you are not familiar with C programming in an IRIX environment. The four lines that begin with the `if (argc != 2) {` test to determine whether you typed a parameter when you ran the program. In other words, if the compiled file were called *ngon*, then you should run it as *ngon 14* or *ngon 24*. The line that contains "`n = atoi(argv[1]);`" converts the parameter from ASCII to integer n. "`atoi`" stands for "ASCII to integer."

The purpose of this example is to draw exactly one n-gon, so there is no real penalty for computing the coordinates of the vertices between `bgnclosedline` and `endclosedline`. If it were necessary to draw the polygon over and over again, the calculated vertices should probably be saved in an array. In real applications that draw objects repeatedly with different viewing parameters, it is usually more efficient to save the coordinates in arrays. You did not do this in the example because it was not worth the effort to save the coordinates for a figure you wanted to draw only once.

2.1.4 Points

The Graphics Library has two high-performance subroutines that allow you to draw a set of points. To draw a set of unconnected points, enter a set of vertices specified between `bgnpoint` and `endpoint`. The system draws each vertex as a one-pixel point on the screen. Below is a sample program that draws a set of unconnected points arranged in a square pattern. The square is 20 pixels wide by 20 pixels high, and the points are spaced 10 pixels apart.

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

main()
{
    long vert[2];
    int i, j;

    prefsize(400, 400);
    winopen("pointpatch");
    color(BLACK);
    clear();
    color(WHITE);
    for (i = 0; i < 20; i++) {
        vert[0] = 100 + 10 * i;  /* load the x coordinate */
        bgnpoint();
        for (j = 0; j < 20; j++) {
                                 /* load the y coordinate */
            vert[1] = 100 + 10 * j;
            v2i(vert);           /* draw the point */
        }
        endpoint();
    }
    sleep(10);
    gexit();
    return 0;
}
```

2.2 Polygons

In the Graphics Library, a polygon is specified by a sequence of distinct vertices: *v1*, *v2*,..., *vn*, that all lie in a plane. You can define the boundary of the polygon by connecting *v1* to *v2*, *v2* to *v3*, and so on, finally connecting *vn* back to *v1*. These connecting segments are called edges. The interior of the polygon is the area inside this region bound by line segments. A polygon is said to be simple if edges intersect only at their common vertices, i.e., the edges cannot cross or touch each other.

A polygon is convex if the line segment joining any two points in the figure is completely contained within the figure. Nonconvex polygons are sometimes called concave. Algorithms that render only convex polygons are much simpler than those that can render both convex and concave polygons.

The Graphics Library and hardware can correctly render any polygon if it is simple, or if it consists of exactly four points. The special case of a bowtie polygon (4-vertex nonsimple polygon) is handled in a hardware-specific manner. The polygon appears either as a bowtie, or as a quadrilateral with a segment missing from one edge (see Figure 2–6).

Some versions of the hardware automatically check for and draw concave polygons correctly, but others do not. The function `concave` guarantees that the system renders concave polygons correctly. On some hardware there is a performance penalty when you use `concave`. If you intend to draw concave polygons, use `concave`, even if your code is running on a machine that automatically does the correct thing. There is a minor penalty for setting the concave flag, but it makes the code portable to other Silicon Graphics machines.

The IRIS draws a polygon as a filled area on the screen. It draws polygons using the same basic syntax it uses for polylines and sets of points—a list of vertex subroutines surrounded by `bgnpolygon` and `endpolygon`. For example, the following program draws a filled blue hexagon on the screen:

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

float hexdata[6][2] = {
    {20.0, 10.0},
    {10.0, 30.0},
    {20.0, 50.0},
    {40.0, 50.0},
    {50.0, 30.0},
    {40.0, 10.0}
};

main()
{
    long i;

    prefposition(100, 500, 100, 500);
    winopen("bluehex");
    color(BLACK);
    clear();
    color(BLUE);
    bgnpolygon();
    for (i = 0; i < 6; i = i+1)
        v2f(hexdata[i]);
    endpolygon();
    sleep(3);
    gexit();
    exit(0);
}
```

Polygons must have fewer than 256 vertices. As it does with closed lines, the Graphics Library software connects the first and the last point; you do not need to repeat the first point.

A model for the procedure for drawing a polygon is:

1. Begin with a list of vertices.

2. Draw a line segment between each vertex and the preceding one.

3. On reaching the last vertex in the list, draw a line from that vertex to the first vertex in the list.

4. Fill the area that this line circumscribes.

There are cases when this procedure does not generate a true polygon, but for simple enclosed areas, this intuitive procedure is sufficient.

Figures 2–1 through 2–6 illustrate some examples of polygons. The heavy black dots represent vertices and the lines represent edges. Figure 2–1 is a convex and simple polygon. Figure 2–2 is a simple, but not convex polygon (a line connecting interior points near the two lowest vertices would go outside the polygon). Figure 2–3 is still simple, but again, not convex.

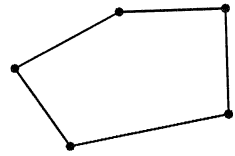


Figure 2–1. Simple Convex Polygon

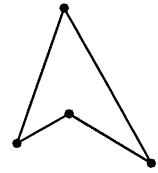


Figure 2–2. Simple Concave Polygon

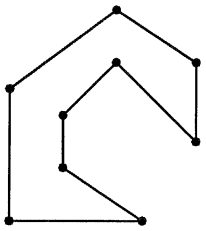


Figure 2–3. Another Simple Concave Polygon

Figures 2–4, 2–5, and 2–6 are not simple; Figure 2–6 illustrates a bowtie polygon. The Graphics Library would correctly render the polygons in Figures 2–1, 2–2, 2–3, and 2–6. The results of rendering Figures 2–4 and 2–5 are unpredictable.

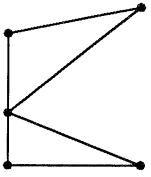


Figure 2–4. Non-simple Polygon

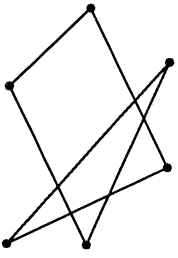


Figure 2–5. Another Non-simple Polygon

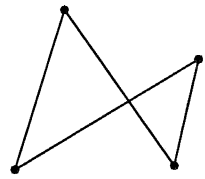


Figure 2–6. "Bowtie" Polygon

Certain distortion problems can arise when viewing a polygon. Sometimes these distortions arise from floating point inaccuracies. But viewing distortions can also arise if the vertices of the polygon were originally specified in three dimensions, and then were transformed and projected to two dimensions (the screen). The only unrecognizable distortion possible for a true polygon (that is, a polygon whose vertices lie in a single plane) is to view it edge on, in which case it collapses to a line (and is not drawn).

However, if the defining vertices for the polygon do not all lie in a plane, the projected polygon on the two-dimensional screen might appear to have duplicate vertices, or crossing edges. The system sometimes creates these not-quite-true polygons when it uses a mesh of polygons to model a curved surface. For most of the surface, the polygons formed by the mesh are nearly flat (true) polygons. However, as the surface twists, the mesh must twist and the view of the mesh might generate bowtie polygons. (This effect is most noticeable at silhouette edges where the mesh curves around to the back of the depicted object.)

The Graphics Library can render the bowties that arise from surface approximating meshes. In most other circumstances, however, the Graphics Library routines for generating polygons generate only true polygons.

2.2.1 Point Sampled Polygons

This section tells exactly which pixels are turned on when the system displays a polygon. You can skip this section on first reading.

To represent a polygon on the screen, the system must turn on a set of pixels. Given a set of coordinates for the vertices of a polygon, there is more than one way to decide which pixels ought to be turned on. The new high-performance subroutines draw point sampled polygons, while the older subroutines (for example, `polf, rect, circle`) draw outlined point sampled polygons. The older subroutines are described in Section 2.8, "Old-Style Drawing."

To illustrate the point sampling method and the reasons for using it, consider drawing two rectangles: rectangle 1 has 2 <= x <= 5 and 1 <= y <= 4, rectangle 2 has 2 <= x <= 5 and 4 <= y <= 6. What pixels should the system turn on in both cases? The most obvious answer is shown in Figure 2–7.

If you draw a figure consisting of the two polygons in Figure 2–7, you would expect them to fit together. Unfortunately, if you draw them both, the pixels on the line y = 4 are drawn twice, once for each polygon. A similar problem would occur if you try to abut a polygon to the right. Normally, this is not a problem, but if the polygons represent a transparent surface, the duplicated edge would be twice as dense, giving the entire surface a spiderweb-like appearance.

Even if the surface is not transparent, there can still be undesired visual effects. If you draw a checkerboard pattern with edges that overlap by exactly one pixel and then redraw it in single buffer mode, the redrawing is visible because the edges of the squares flicker from one color to the other, even though the final second image is identical to the first. See Chapter 6, "Animation," for more about single buffer mode.

The system resolves these problems using point sampled polygons. The model is this: ideal mathematical lines (no thickness) connect the vertices. The system draws any pixel whose center lies inside the mathematically precise polygon; it does not draw a pixel if its center lies outside the polygon. Pixels whose centers lie exactly on the mathematical line segments or vertices are filled in in a hardware-dependent manner that attempts to avoid both multiple fills and gaps at the boundaries of adjacent polygons. All that is guaranteed about this algorithm is that pixels on the left and bottom (and not the right and top) of a screen-aligned rectangle drawn on exact pixel centers are filled.

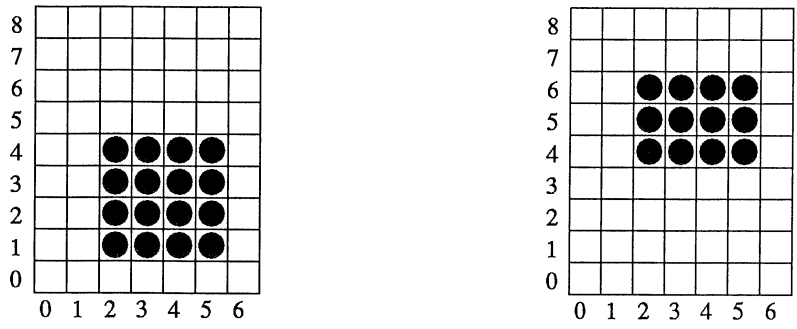


Figure 2–7. Non-Point Sampled Polygons

This definition effectively eliminates the duplication of pixels from the right and top edges of the polygon, but adjacent polygons can fill those pixels. Figure 2–8 shows point sampled versions of the two rectangles above.

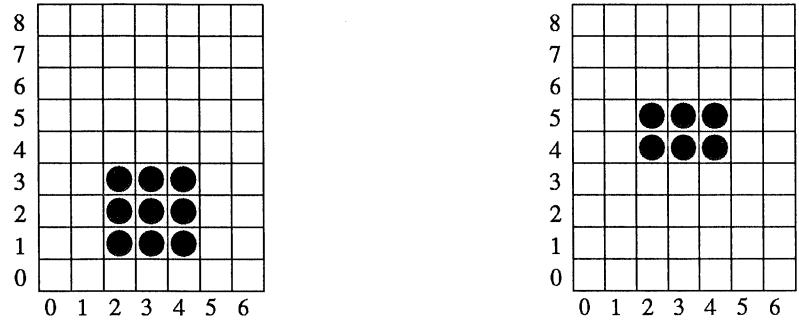


Figure 2–8. Point Sampled Polygons

Another advantage of a point sampled polygon without an outline is that the drawn area of the polygon is much closer to the actual mathematical area of the polygon. In the examples above, the drawn areas correspond exactly to the true areas of the polygons. In nonrectangular polygons, the drawn area of the polygon cannot be exact, but the drawn area of the no-outline point sampled polygon is closer to the true area of the polygon than the area drawn by the older outlined model.

Figure 2–9 illustrates the pixels that would be turned on in a point sampled representation of the polygon that connects the vertices (1,1) (1,4) (5,6) and (5,1). The darkened pixels would be drawn. The pixels at (1, 4), (3, 5), (5,6), (5,5), (5,4), (5,3), (5,2), and (5,1) all lie mathematically on the boundary of the polygon but are not drawn because they are on the upper or right edge.

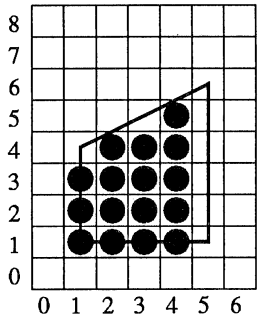


Figure 2–9. Point Sampled Polygon

As mathematic entities, lines have no thickness. However, to represent a line on the screen, the system assumes a thickness of exactly one pixel. When you scale an object composed of lines, the lines behave differently from polygons. No matter how much a transformation magnifies or reduces an object composed of lines, the representation of the line remains one pixel thick. If a line is drawn around a point sampled polygon, it fills in the pixels at the upper and right-hand edges. For compatibility, the older subroutines such as `polf`, `rect`, `circle`, effectively do this, i.e., draw a line around the point sampled version. See Section 2.8, "Old-Style Drawing," for information about the older subroutines.

Anomalies can occur in the display of very thin filled polygons. For example, consider the point sampled rendition of the triangle connecting the points (1,1), (2,3), and (12,7). It is apparently riddled with holes, as illustrated in Figure 2–10, but if adjacent polygons that share the vertices are drawn, all the pixels are eventually filled.

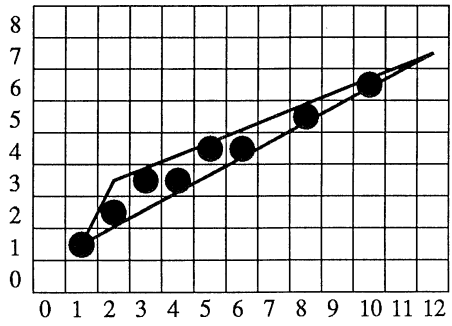


Figure 2–10. Point Sampling Anomaly

2.2.2 Meshes

This section covers an advanced topic; on the first reading, you can skip it. It concerns the specification of geometric figures constructed entirely of adjacent triangles, i.e., triangular meshes. Triangular meshes provide a very efficient way to specify three-dimensional objects that are composed of triangular faces.

A triangular mesh is a set of triangles formed from a series of points. In Figure 2–11, the seven vertices form five triangles (123, 324, 345, 546, 567). Points 1 and 7 appear in one triangle; points 2 and 6 appear in two triangles, and all the rest appear in all three. In a longer sequence, a higher percentage of the points would be used three times. If the mesh in Figure 2–11 is drawn as five separate triangles, many of the points are transformed multiple times (in fact, transformation to screen coordinates occurs 15 times, although there are only 7 points). The triangular mesh primitive provides a more efficient way to display sequences of triangles like those shown in Figure 2–11.

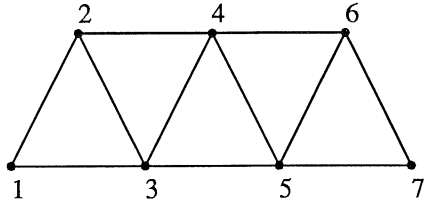


Figure 2–11. Simple Triangle Mesh

Figure 2–11 illustrates the simplest case. It uses the sequence {bgntmesh(); v(1); v(2); v(3); v(4); v(5); v(6); v(7); endtmesh();}, where v(i) stands for any vertex subroutine with the coordinates of the i-th point. As a result, the pipeline accepts (and transforms) points 1 and 2. When point 3 arrives, it is transformed and the system draws the triangle 123. Then point 3 replaces point 1 (so the pipeline now remembers points 2 and 3), and when point 4 arrives, triangle 324 is drawn, and point 4 replaces point 2.

This sequence continues. Each time a new point is sent, the system draws a triangle containing it and the two retained points. The oldest retained point is then discarded, and is replaced by the new point. The sequence ends when `endtmesh` is sent.

Figure 2–12 illustrates a more complex situation. The first six triangles (123, 234, 345, 456, 567, 678) could be drawn as before, but if nothing is done, the arrival of point 9 would cause triangle 789 to be drawn, not triangle 689 as desired. To draw meshes like the one in Figure 2–12, you must examine more closely the mechanism the geometry hardware uses to retain points.

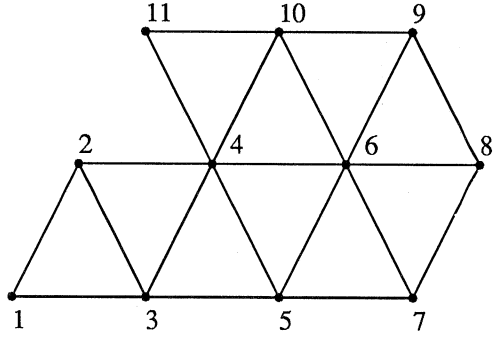


Figure 2–12. swaptmesh Example

The pipeline maintains two previous vertices together with a pointer that points to one or the other of them while drawing a triangle mesh. When a new vertex arrives, a triangle is drawn using all three vertices, and then the new vertex replaces the one pointed to by the pointer. The pointer is then changed to point to the other retained vertex. Thus if nothing special is done, the discarded vertex alternates, drawing a picture like the one in Figure 2–13.

Below is an illustration of what happens internally to draw Figure 2–11.

Initial state:	After vertex1:	After vertex2:
P -> R1 = junk1	R1 = vert1	P -> R1 = vert1
R2 = junk2	P-> R2 = junk2	R2 = vert2

When vertex3 arrives, triangle 123 is drawn, and the state is:

R1 = vert3

P ->R2 = vert2

The next few states are:

After drawing 324:	After 345:	After 546:
P -> R1 = vert3	R1 = vert5	P -> R1 = vert5
R2 = vert4	P -> R2 = vert4	R2 = vert6

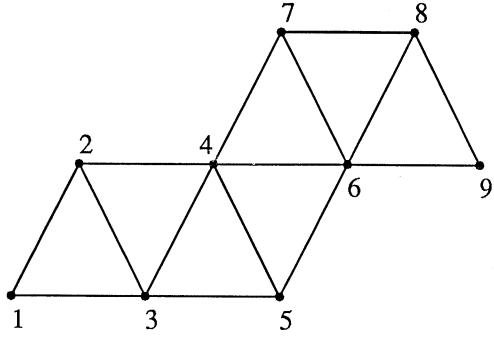


Figure 2–13. Another swaptmesh Example

The Graphics Library contains another subroutine, `swaptmesh`, whose only effect is to swap the pointer to the other retained vertex. The following sequence draws the mesh in Figure 2–13.

```
bgntmesh();
 v(1);
 v(2);
 v(3);
 v(4);
 v(5);
 v(6);
 v(7);
swaptmesh();
 v(8);
swaptmesh();
 v(9);
swaptmesh();
 v(10);
 v(4);
 v(11);
endtmesh();
```

Here is what is happening internally:

After vertex7:	After swaptmesh:	After vertex8:
R1 = vert7	P -> R1 = vert7	R1 = vert8
P -> R2 = vert6	R2 = vert6	P -> R2 = vert6
After swaptmesh:	After vertex9:	After swaptmesh:
P -> R1 = vert8	R1 = vert9	P -> R1 = vert9
R2 = vert6	P -> R2 = vert6	R2 = vert6
After vertex10:	After vertex4:	After vertex11:
R1 = vert10	P -> R1 = vert10	R1 = vert11
P -> R2 = vert6	R2 = vert4	P -> R2 = vert4

Without going into such detail, here is the sequence that would draw Figure 2–13:

```
bgntmesh();
  v(1);
  v(2);
  v(3);
  v(4);
  v(5);
swaptmesh();
  v(6);
  v(7);
swaptmesh();
  v(8);
  v(9);
endtmesh();
```

The following sample program draws a three-dimensional octahedron (8-sided regular polyhedron) using the mesh primitive. Since meshes in two dimensions are of little use, the example is three-dimensional. Knowing how to create a three-dimensional mesh is quite useful; however, it uses a number of routines that are covered in later chapters including three-dimensional rotations, hidden surface removal, smooth (double buffered) motion, and a different color mode.

For the purpose of illustration, the following program defines a drawing subroutine called `drawoctahedron` that uses the mesh subroutine.

The `shademodel` statement affects the behavior of triangular mesh in the following ways:

- `shademodel(FLAT)` means that each triangle is filled with the color that was current when the last vertex routine executed (this is called the "provoking vertex").

- `shademodel(GOURAUD)` means that each triangle is Gouraud-shaded using the colors assigned to all three vertices.

The `cpack` subroutines set vertex colors, so ignore them if you are studying the program to understand the logic of mesh drawing. All the rotation and hidden surface removal are handled in the `main()` routine. The calculations of rotation angles simply cause the octahedron to tumble in an interesting way.

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

float octdata[6][3] = {
    {1.0, 0.0, 0.0},
    {0.0, 1.0, 0.0},
    {0.0, 0.0, 1.0},
    {-1.0, 0.0, 0.0},
    {0.0, -1.0, 0.0},
    {0.0, 0.0, -1.0}
};

drawoctahedron()
{
    bgntmesh();
    cpack(0xff0000);        /* color blue */
    v3f(octdata[0]);
    cpack(0x00ff00);        /* color green */
    v3f(octdata[1]);
    swaptmesh();
    cpack(0x0000ff);        /* color red */
    v3f(octdata[2]);
    swaptmesh();
    cpack(0xffff00);        /* color cyan */
    v3f(octdata[4]);
    swaptmesh();
    cpack(0xffffff);        /* color white */
    v3f(octdata[5]);
    swaptmesh();
    cpack(0x00ff00);        /* color green */
    v3f(octdata[1]);
    cpack(0xff00ff);        /* color magenta */
    v3f(octdata[3]);
    cpack(0x0000ff);        /* color red */
    v3f(octdata[2]);
    swaptmesh();
    cpack(0xffff00);        /* color cyan */
    v3f(octdata[4]);
    swaptmesh();
```

```
    cpack(0xffffff);        /* color white */
    v3f(octdata[5]);
    swaptmesh();
    cpack(0x00ff00);        /* color green */
    v3f(octdata[1]);
    endtmesh();
}

main()
{
    long iang, jang, kang;
    long exitcounter;

    prefposition(100, 500, 100, 500);
    winopen("octahedron");
    ortho(-2.0, 2.0, -2.0, 2.0, -2.0, 2.0);
    zbuffer(TRUE);   /*hidden surfaces removed
                       with z buffer*/
    doublebuffer();  /* for smooth motion */
    RGBmode();       /* direct color mode */
    gconfig();       /* reconfigure for RGBmode and
                       doublebuffer */

    for (exitcounter=0; exitcounter < 1000; exitcounter++)
    {
        pushmatrix();      /*save viewing transformation*/
        rotate(iang, 'x'); /*rotate by iang about x axis*/
        rotate(jang, 'y'); /*rotate by jang about y axis*/
        rotate(kang, 'z'); /*rotate by kang about z axis*/
        iang += 10;
        jang += 13;
        if (iang + jang > 3000) kang += 17;
        if (iang > 3600) iang -= 3600;
        if (jang > 3600) jang -= 3600;
        if (kang > 3600) kang -= 3600;
        cpack(0);        /* color black */
        clear();
        zclear();        /* clear the z buffer */
        drawoctahedron();
        swapbuffers();   /* show completed drawing */
        popmatrix();     /* restore viewing transformation */
    }
}
```

2.2.3 Quadrilateral Strips

In addition to the triangular mesh, the Graphics Library also supports quadrilateral strips. Quadrilateral strips are similar in many ways to triangular meshes, but might be better suited for the representation of shapes that are fundamentally quadrilateral rather than triangular in nature.

bgnqstrip and endqstrip

`bgnqstrip` and `endqstrip` interpret vertex (`v`) subroutines as quadrilateral strip vertices.

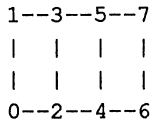
```
void bgnqstrip()
void endqstrip()
```

These two statements delimit a sequence of vertex commands that specify a strip of connected quadrilaterals. The `bgnqstrip` and `endqstrip` commands must surround an even number of vertex commands that is four or greater, and is unbounded. Filling results are undefined if these conditions are not met.

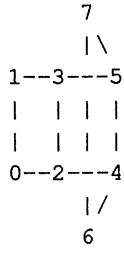
Vertices specified after `bgnqstrip` and before `endqstrip` form a sequence of quadrilaterals. You cannot alter the replacement algorithm, since there is no quadrilateral equivalent to the `swaptmesh` command. For example, the sequence:

```
bgnqstrip()
    v3f(vert0)
    v3f(vert1)
    v3f(vert2)
    v3f(vert3)
    v3f(vert4)
    v3f(vert5)
    v3f(vert6)
    v3f(vert7)
endqstrip()
```

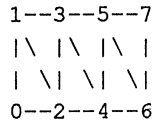
draws three quadrilaterals: (0,1,3,2), (2,3,5,4), and (4,5,7,6).



The system determines quadrilateral direction (for purposes of backface elimination and two-sided lighting) for each quadrilateral as though each quadrilateral were an independent polygon with vertex order (*n*, *n*+1, *n*+3, *n*+2). Thus the three quadrilaterals in the example above are all backfacing, because vertices (0,1,3,2), (2,3,5,4), and (4,5,7,6) all have clockwise screen rotation. In the following example, however, quadrilaterals (0,1,3,2) and (2,3,5,4) are backfacing, but quadrilateral (4,5,7,6) is frontfacing (its screen rotation is counter-clockwise).



Note that the vertex order required by quad strips matches the order required for "equivalent" triangle meshes. Thus the first example vertex sequence produces:



when bounded by `bgntmesh()` and `endtmesh()` calls. In general, quadrilateral data looks better when drawn with `qstrip` than with `tmesh`. This is because Gouraud-shading calculations operate on the original quadrilateral data, rather than on the decomposed triangles.

There is no maximum number of vertices that can be specified between `bgnqstrip` and `endqstrip`. If the number is odd, however, the result is undefined.

Note: The IRIS-4D VGX graphics hardware uses vertex normals to determine how to decompose quadrilaterals into triangles during scan conversion. If you do not specify vertex normals, or (equivalently) if the four vertices share the same normal, the selected decomposition matches that of the equivalent triangle mesh.

2.2.4 Controlling Polygon Rendering

The `polymode` statement lets you specify how the system renders polygons. This statement controls polygons created with triangular mesh or quadrilateral strips as well as explicit polygons (that is, polygons created inside a `bgn/end` loop). Note that this command does not affect the transformation or clipping of polygon vertices. Only polygons generated by `arcf`, `cirf`, `polf`, `rectf`, `splf`, `bgn/end polygon`, `bgn/end tmesh`, `bgn/end qstrip`, and by the NURBS surface software are affected.

```
void polymode(mode)
long mode;
```

mode can be one of the following symbols:

PYM_POINT	draw only points at each vertex
PYM_LINE	draw lines from vertex to vertex
PYM_FILL	fill the polygon interior
PYM_HOLLOW	fill only interior pixels at the boundaries

PYM_POINT and PYM_LINE draw points and lines consistent with all applicable point and line modes. Thus antialiasing (`pntsmooth` and `linesmooth`) as well as `linewidth` and `linestipple` are significant. PYM_FILL is the standard fill operation that was previously the only option.

Polygons drawn in PYM_LINE mode clip differently from closed lines: the PYM_LINE polygon always clips to a closed line, with line segments generated along the edges of the clip planes, usually the viewport.

PYM_HOLLOW supports a special kind of polygon fill with the following properties:

- Only pixels on the polygon edge are filled. These pixels form a single-width line (regardless of the current linewidth) around the inner perimeter of the polygon.
- Only pixels that would have been filled (PYM_FILL) are changed (i.e. the outline does not extend beyond the exact polygon boundaries).
- Pixels that are changed take the exact color and depth values they would have had the polygon been filled.

Because their pixel depth values are exact, hollow polygons can be composed with filled polygons accurately. Both hidden-line and scribed-surface renderings can be done taking advantage of this fact.

Not all IRIS models support polymode. Use `getgdesc` with the GD_POLYMODE argument to determine whether polymode is supported. The IRIS-4D VGX requires special setup to support PYM_HOLLOW. Refer to the manual page *polymode*(3G) for details.

2.3 High-Level Subroutines

Using the high-performance drawing subroutines described in the last section, it is possible to draw any of the primitive geometric figures in the Graphics Library (except curves and surfaces, which are covered in Chapter 14). However, because rectangles, circles, and arcs are drawn so often, the Graphics Library provides subroutines to draw these objects.

In this section and the next, most of the subroutine names follow a pattern. If the geometric figures they draw are filled (i.e., are polygons), the root name has an *f* appended to it. There is no *f* if the figure is unfilled. For example, `rect` draws a rectangular outline, while `rectf` draws a filled (solid) rectangle. The arguments to the subroutines can be short integers (16 bits), long integers (32 bits), or floating point numbers (32 bits). Floating point is the default, but if the argument type is short integer, there is an *s* suffix. If the argument type is a long integer, the subroutine name takes an *i* suffix. As with the vertex subroutine, only long integers in the range of -2^{23} and $2^{23}-1$ are converted correctly. Treat long integers as 24-bit 2s-complement numbers sign-extended to 32 bits.

For example, the rectangle subroutine has six forms: short integer filled, long integer filled, floating point filled, short integer unfilled, long integer unfilled, and floating point unfilled. The names for these six are, respectively, `rectfs`, `rectfi`, `rectf`, `rects`, `recti`, and `rect`.

2.3.1 Rectangles

The Graphics Library provides two types of rectangle subroutines—filled and unfilled. Filled rectangles are just rectangular polygons, and unfilled rectangles are rectangular outlines. Only the *x* and *y* coordinates of the corners of the rectangle are given, and the *z* coordinate is assumed to be zero. The rectangle is assumed to be aligned with the *x* and *y* axes.

Table 2–2 lists the six different forms of the rectangle subroutine.

	Filled	Unfilled
16-bit integer	rectfs	rects
32-bit integer	rectfi	recti
32-bit float	rectf	rect

Table 2–2. The Rectangle Subroutine

The arguments to all six rectangle subroutines are the same: `rect(x1, y1, x2, y2)`. The point *(x1, y1)* is one corner of the rectangle and *(x2, y2)* is the opposite corner. Because the rectangle is assumed to be aligned with the axes, the coordinates of the other corners would be *(x1, y2)* and *(y1, x2)*.

Rectangles can undergo three-dimensional geometric transformations as described in Chapter 7, and the resulting figure need not appear to be a rectangle. (For example, imagine rotating the rectangle about the *x* axis so that one end is farther from you and then viewing it in perspective. On the screen, the rotated rectangle would appear to be a trapezoid.)

It is important to understand, however, that although rectangles created with `rectf` or its variants can be transformed by the statements described in Chapter 7, "Coordinate Transformations," the *z* coordinates of such rectangles (and circles, arcs, and other 2-D figures) remain zero, and the apparent rotation or translation takes place because of manipulations to the underlying transformation matrix. If you wish to build a composite of different rectangular shapes (for instance, a 3-D cube) that is to be part of a 3-D model, the correct way is to use the 3-D drawing functions `bgnpolygon()` and `endpolygon()`.

The following sample program draws a chess board with black and white squares on a green background using the rectangle subroutine. In addition (to demonstrate the unfilled rectangle subroutines), there is a red line outlining the board.

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

main()
{
    long i, j;

    prefposition(100, 500, 100, 500);
    winopen("chessboard");
    color(GREEN);
    clear();
    for (i = 0; i < 8; i = i+1)
    for (j = 0; j < 8; j = j+1) {
        if (odd(i+j))
            color(WHITE);
        else
            color(BLACK);
        rectfi(100 + i*25,
            100 + j*25,
            124 + i*25,
            124 + j*25);
        }
    color(RED);
    recti(97, 97, 302, 302);
    sleep(3);
    gexit();
    exit(0);
}

odd(n) /* returns 1 if n is odd; 0 otherwise. */
long n;
{
    return n&1;
}
```

Screen Boxes

Screen boxes are a subclass of rectangles. They are always 2-D only and are always aligned with the screen coordinates. Two fast primitives exist in the Graphics Library for drawing screen boxes: `sbox` and `sboxf`. As with `rect` and `rectf`, the "f" signifies that the screen box is filled with the current color and pattern.

All screen box commands expect four arguments:

x1 *x* coordinate of one corner of the box

y1 *y* coordinate of one corner of the box

x2 *x* coordinate of the opposite corner of the box

y2 *y* coordinate of the opposite corner of the box

The screen box drawing commands (Table 2–3) fill in the rectangle given these diagonal corner coordinates. The `sbox` statements draw 2-D, screen-aligned rectangles using the current color, writemask, and linestyle. The `sboxf` statements draw filled 2-D, screen-aligned rectangles using the current color, writemask, and pattern.

	Filled	Unfilled
16-bit integer	sboxfs	sboxs
32-bit integer	sboxfi	sboxi
32-bit floating point	sboxf	sbox

Table 2–3. Screen Box Commands

You cannot use lighting, backfacing, depth-cueing, z-buffering, Gouraud shading, or alpha blending with the `sbox` or `sboxf` commands.

2.3.2 Circles

Like rectangles, circles are two-dimensional figures, and lie in the *x*-*y* plane, with *z* coordinates equal to zero. If they are viewed at an angle, circles appear to be ellipses.

The arguments for the circle subroutines include the center point *(x, y)* and the radius. Like rectangles, circles are either filled or unfilled, and the center coordinates and radius are specified in integers, short integers, or floating point numbers. Table 2–4 lists the six different circle subroutines.

	Filled	Unfilled
16-bit integer	circfs	circs
32-bit integer	circfi	circi
32-bit floating point	circf	circ

Table 2–4. The Circle Subroutine

The parameters to all six subroutines are the same: `circ(x, y, radius)`. Circles are drawn with 80 equally spaced vertices, either as a closed line (for unfilled circles) or as a polygon (for filled circles). If your application draws many tiny circles, it is a good idea to write a circle primitive that uses fewer line segments, and that can therefore be drawn much more quickly. A similar problem can arise for very large circles—if they are magnified enough, you can easily see the individual straight line segments. However, circles drawn with 80 segments look reasonably good over a wide range of sizes.

The following sample program draws an archery target using filled circles:

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

main()
{
    prefposition(100, 500, 100, 500);
    winopen("bullseye");
    ortho2(-1.0, 1.0, -1.0, 1.0);
    color(BLACK);
    clear();
    color(GREEN);
    circf(0.0, 0.0, 0.9);
    color(YELLOW);
    circf(0.0, 0.0, 0.7);
    color(BLUE);
    circf(0.0, 0.0, 0.5);
    color(CYAN);
    circf(0.0, 0.0, 0.3);
    color(RED);
    circf(0.0, 0.0, 0.1);
    sleep(3)
    gexit();
    exit(0);
}
```

2.3.3 Arcs

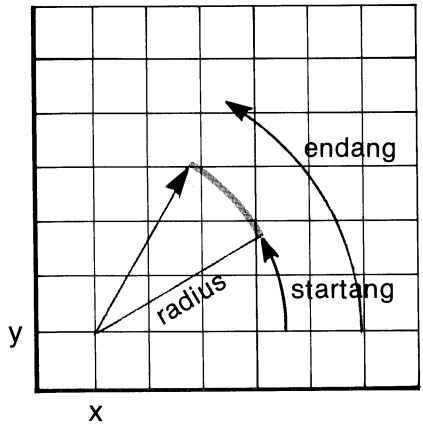
Arcs are also two-dimensional figures, and like circles and rectangles, they are assumed to lie in the plane z = 0. When viewed at an angle, arcs appear to be segments of ellipses. Arcs can be either filled or unfilled. Unfilled arcs are simply segments of circles, while filled arcs look like sections of pie (Figure 2–14).

Arcs are defined by a center *(x, y)*, a radius, a starting angle, and an ending angle. The angles are measured from the positive *x* axis in a counterclockwise direction. Negative angles are measured clockwise. Both angles are expressed as integers in tenths of degrees, so a 90-degree angle would be expressed as 900.

An arc is always drawn counterclockwise from the starting angle to the ending angle, so if `startang` = 0 and `endang` = 100, a 10 degree arc would be drawn. If the starting angle is 100 and ending angle is 0, a 350 degree arc is drawn.

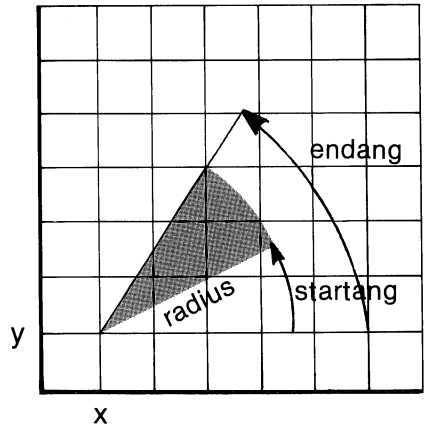
The circular portions of the arcs drawn are approximated by straight lines, and a full 360- degree arc consists of 80 segments. If your application draws many tiny arcs, it is a good idea to write an arcs primitive that uses fewer line segments, and that can therefore be drawn much more quickly. A similar problem can arise for very large arcs—if they are magnified enough, you can easily see the individual straight line segments. However, arcs drawn with 80 segments look reasonably good over a wide range of sizes.

(a)



arci(x,y,radius,startang,endang);

(b)



arcfi(x,y,radius,startang,endang);

A center point, radius, start angle, and end angle define circular arcs. They are drawn counterclockwise in the x-y plane, with angles measured from the x-axis.

Figure 2-14. Arcs

Arcs subroutines come in the same six forms as subroutines for circles and rectangles (Table 2–5):

	Filled	Unfilled
16-bit integer	arcfs	arcs
32–bit integer	arcfi	arci
32-bit float	arcf	arc

Table 2–5. The Arc Subroutine

The order of arguments for all six subroutines is given by: `arc(x, y, radius, startang, endang)`. The following sample program draws a pie chart using filled arcs:

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

main()
{
    prefposition(100, 500, 100, 500);
    winopen("piechart");
    ortho2(-1.0, 1.0, -1.0, 1.0);
    color(BLACK);
    clear();
    color(RED);
    arcf(0.0, 0.0, 0.9, 0, 800);
    color(GREEN);
    arcf(0.0, 0.0, 0.9, 800, 1200);
    color(YELLOW);
    arcf(0.0, 0.0, 0.9, 1200, 2200);
    color(MAGENTA);
    arcf(0.0, 0.0, 0.9, 2200, 3400);
    color(BLUE);
    arcf(0.0, 0.0, 0.9, 3400, 0);
    sleep(3)
    gexit();
    exit(0);
}
```

2.4 Old-Style Drawing

The architecture of Silicon Graphics' earlier systems was tuned to a different set of subroutines for drawing points, lines, and polygons. For compatibility, all of the earlier subroutines are still in the Graphics Library. In most cases, the internals of these earlier subroutines have been rewritten to use the new high-performance subroutines. However, to guarantee that you get the optimal performance of the new programs, use the subroutines described at the beginning of this chapter.

Except for polygons, the figures drawn by the old-style subroutines are the same as those drawn by the high-performance ones. For example, points are drawn as a single pixel. However, the earlier subroutines did not draw point sampled polygons. They effectively drew point sampled polygons with lines connecting the vertices. For compatibility, the old polygon subroutines draw point sampled polygons with an outline, so they appear exactly the way they did before. For many polygons, the drawing time is increased when both the polygon and its outline are drawn.

In most cases, absolute compatibility with the old polygon filling style is not required, so there is a subroutine, `glcompat()`, that you can use to turn off outlining for the old-style subroutines. You can significantly increase polygon drawing performance for old code by turning off the compatibility mode. `glcompat()` takes two arguments. The first is the compatibility mode to be altered, and the second is the value to which it is set. To turn off polygon outlining, use:

```
glcompat(GLC_OLDPOLYGON, 0);
```

The default `GLC_OLDPOLYGON` value is 1 (outlining is turned on).

Note that the subroutines that draw rectangles, circles, and arcs all actually draw polygons, so `glcompat` can turn outlining off or on for all four of these types of filled figures.

The naming conventions for the rest of the subroutines in this chapter are similar to those used by the arc, circle, and rectangle subroutines. However, since the remaining subroutines are usually three-dimensional, they come in two-dimensional and three-dimensional versions. As with arc, circle, and rectangle, the two-dimensional versions are assumed to lie in the plane $z = 0$, but those figures can be transformed out of that plane by the various transformation and viewing subroutines discussed in Chapter 7.

The naming convention assumes that most subroutines are three-dimensional, so, for example, the point subroutine `pnt` is the three-dimensional version, and `pnt2` requires no *z* component to its arguments.

2.4.1 Current Graphics Position

In the new architecture, the graphical figures are sent together—a set of points, a polyline, and a polygon are sent bracketed by a `bgn`*<type>* and an `end`*<type>* subroutine. The rendering of the figure might not start until the `end`*<type>* arrives.

In older systems, points were sent as individual subroutines, lines as a series of move and draw subroutines, and polygons as a polygon move, followed by polygon draw subroutines, and finally a polygon close subroutine.

Between the subroutines drawing polylines or polygons, the system maintained a current graphics position. Each draw subroutine draws from the current graphics position to the point specified by `draw`. The current graphics position is then set to the new point.

Similarly, the current graphics position is used by polygon subroutines discussed in the next sections.

getgpos

The system automatically maintains the current graphics position, so very few applications need to access it directly, although `getgpos` returns the current graphics position. Its arguments include four pointers to floating point numbers in which the homogeneous coordinates of the current transformed point are returned. The returned values are in clip coordinates (see Section 7).

For compatibility, the current graphics position is maintained in exactly the same way for all the graphics subroutines listed in the rest of this chapter. All the graphics subroutines mentioned up to now do not depend on the current graphics position, and in fact, leave it in an unpredictable state.

```
void getgpos(fx, fy, fz, fw)
Coord *fx, *fy, *fz, *fw;
```

2.4.2 Points

There are six versions of the point subroutine (Table 2–6).

	2-D	3-D
16-bit integer	pnt2s	pnts
32-bit integer	pnt2i	pnti
32-bit float	pnt2	pnt

Table 2–6. The pnt Subroutine

The argument lists are: `pnt2(x, y)` and `pnt(x, y, z)`. In addition to drawing a point, `pnt` updates the current graphics position to its location.

The following program draws 100 points in a square area of the window:

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

main()
{
    int i, j;

    prefposition(100, 500, 100, 500);
    winopen("pointsquare");
    color(BLACK);
    clear();
    color(BLUE);
    for (i = 0; i < 10; i = i+1)
        for (j = 0; j < 10; j = j+1)
            pnti(i*5, j*5, 0);
    sleep(3);
    gexit();
    exit(0);
}
```

2.4.3 Lines

Lines can be drawn using two subroutines: `move` and `draw`.

The `move` subroutine sets the current graphics position to the specified vertex, and `draw` draws from the current graphics position to the specified point and then updates the current graphics position to that vertex. The arguments and types of the move and draw subroutines are the same as for the point subroutines. Table 2–7 is a complete list of the move and draw subroutines.

	2–D	3-D
16-bit integer	move2s	moves
32-bit integer	move2i	movei
32-bit float	move2	move
16-bit integer	draw2s	draws
32-bit integer	draw2i	drawi
32-bit float	draw2	draw

Table 2–7. The move and draw Subroutines

This sample program draws the outline of a blue box on the screen using the move and draw subroutines:

```
#include <gl/gl.h>
]
main()
{
    prefposition(100, 500, 100, 500);
    winopen("bluebox");
    color(BLACK);
    clear();
    color(BLUE);
    move2i(200, 200);
    draw2i(200, 300);
    draw2i(300, 300);
    draw2i(300, 200);
    draw2i(200, 200);
    sleep(3);
    gexit();
    exit(0);
}
```

2.4.4 Polygons

The old-style subroutines that draw filled polygons corresponding to the move and draw subroutines are `pmv` and `pdr`. Table 2–8 is a complete list of the filled polygon subroutines.

	2-D	3-D
16-bit integer	pmv2s	pmvs
32-bit integer	pmv2i	pmvi
32-bit float	pmv2	pmv
16-bit integer	pdr2s	pdrs
32-bit integer	pdr2i	pdri
32-bit float	pdr2	pdr

Table 2–8. The Filled Polygon Subroutines

A polygon is specified by a `pmv` to locate the first point on the boundary, then a sequence of `pdr` subroutines for each additional vertex, and finally a `pclos` to close and fill the polygon. The `pclos` subroutine has no arguments; all the other subroutines take either two or three arguments of the appropriate type.

The following sample program draws a filled blue polygon:

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

main()
{
    prefposition(100, 500, 100, 500);
    winopen("bluebox");
    color(BLACK);
    clear();
    color(BLUE);
    pmv2i(200, 200);
    pdr2i(200, 300);
    pdr2i(300, 300);
    pdr2i(300, 200);
    pclos();
    sleep(3);
    gexit();
    exit(0);
}
```

Note that `pclos` connects back to the original starting point.

Finally, the Graphics Library has two sets of subroutines that take arrays of vertex coordinates and draw filled and unfilled polygons. These subroutines draw exactly the same figures as the move and draw (or polygon move and polygon draw) subroutines, but they are often more convenient to use.

Filled polygons are drawn by `polf`, and polygon outlines are drawn by `poly`. Table 2–9 is a complete list of the polygon and filled polygon subroutines.

	2-D	3-D
16-bit integer	poly2s	polys
32-bit integer	poly2i	polyi
32-bit float	poly2	poly
16-bit integer	polf2s	polfs
32-bit integer	polf2i	polfi
32-bit float	polf2	polf

Table 2–9. The Polygon and Filled Polygon Subroutines

Both the `polf` and the `poly` subroutines take two arguments. The first argument, *n*, is the number of vertices in the polygon, and the second is a two-dimensional array containing the coordinates.

This program draws a hexagon using `polf` :

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

long parray[6][2] = {{200,100},{100,300},{200,500},
                     {400,500},{500,300},{400,100}};

main()
{
    prefposition(100, 600, 100, 600);
    winopen("hexagon");
    color(BLACK);
    clear();
    color(GREEN);
    polf2i(6, parray);
    sleep(3);
    gexit();
    exit(0);
}
```

2.5 Linestyles

Linestyle is the term used to describe the way the system draws lines on the screen. It represents a 16-bit pattern on the screen. The system runs this pattern repeatedly to determine which pixels in a 16-pixel line segment it must color. For example, the linestyle 0xFFFF draws a solid line; 0xF0F0 draws a dashed line; and 0x8888 draws a dotted line. The least significant bit of the pattern is the mask for the first pixel of the line and every sixteenth pixel thereafter.

deflinestyle

`deflinestyle` defines a linestyle. Its arguments specify an index into a table (*n*), which stores linestyles and a 16-bit linestyle pattern (*ls*). There are 2^{16} possible linestyle patterns. By default, index 0 contains linestyle 0xFFFF, which draws solid lines. You cannot redefine the linestyle at index 0.

To replace a linestyle, specify the index of the old linestyle in place of the new one.

```
void deflinestyle(n, ls)
short n;
Linestyle ls;
```

setlinestyle

There is always a current linestyle; the system uses it to draw lines and to outline rectangles, polygons, circles, and arcs. Linestyle 0 is the default linestyle. Use `setlinestyle` to select another linestyle. Its argument, *index*, is an index into the linestyle table built by calls to `deflinestyle`.

```
void setlinestyle(index)
short index;
```

2.5.1 Modifying the Linestyle Pattern

Two routines modify the application of the linestyle pattern: `lsrepeat` and `linewidth`. You can get the current values for these attributes using `getstyle`, `getlsrepeat`, and `getlwidth`.

lsrepeat

Use `lsrepeat` to create linestyles that are longer than 16 bits. `lsrepeat` multiplies each bit in the pattern by *factor*. Consequently, each 0 in the linestyle pattern becomes a series of *factor* x 0, and each 1 becomes a series of *factor* x 1. For example, if the pattern is 0xFE00 and *factor*=1, the linestyle is nine bits off followed by seven bits on; f *factor*=3, the linestyle is 27 bits off followed by 21 bits on.

```
void lsrepeat(factor)
long factor;
```

linewidth

`linewidth` specifies the width of a line. The system measures the width in pixels along the *x* axis or along the *y* axis. It defines the width of a line as the number of pixels along the axis having the smallest difference between the endpoints of the line.

```
void linewidth(n)
short n;
```

getlstyle

`getlstyle` returns the index of the current linestyle.

```
long getlstyle()
```

getlsrepeat

`getlsrepeat` returns the factor (integer) by which the linestyle is multiplied for patterns that are longer than 16 bits.

```
long getlsrepeat()
```

getlwidth

`getlwidth` returns the current linewidth in pixels.

```
long getlwidth()
```

2.6 Patterns

You can fill rectangles, polygons, and arcs with arbitrary patterns. A pattern is an array of short integers that defines a rectangular pixel array. The pattern controls which pixels the system colors when it draws filled polygons. The system aligns the pattern to the lower-left corner of the screen, rather than to the filled shape, so that the pattern appears continuous over large areas.

defpattern

`defpattern` defines patterns. Its arguments specify an index into a table of patterns (*n*), a size (*size*), and an array of short integers (*mask*). A pattern can be 16x16 or 32x32. The origin of the pattern is the lower-left corner of the screen. You define the bottom row first. Specify each row of a 16x16 pattern with a single short; specify each row of a 32x32 pattern with two shorts, first the left 16 bits, then the right. Bit 0 of each short is the rightmost bit of its respective position in the row.

You cannot redefine the pattern at index 0.

```
void defpattern(n, size, mask)
short n, size;
short *mask;
```

setpattern

`setpattern` selects which defined pattern the system uses `defpattern` provides an index that you use as the argument for `setpattern`. Pattern 0 is the default pattern.

```
void setpattern(index)
short index;
```

getpattern

`getpattern` returns the index of the current pattern.

```
long getpattern()
```

3. Characters and Fonts

Chapter 2 describes how to draw geometric figures such as points, lines, and polygons. This chapter tells one way to display textual information.

The IRIS-4D supports the rapid display of rasterized characters in multiple fonts. The fonts can be fixed or variable pitch, and can come in different point sizes. You can design and use your own fonts or make use of the default fonts supplied with the system.

This chapter describes the subroutines that position and draw characters, define fonts, and determine information about the currently defined font.

The NeWS™ font manager described in the 4Sight documentation offers a more flexible way to display text in a program than the method presented in this chapter. Since the NeWS font manager used by 4Sight is also supported in the X Window System™ implementation on IRIS-4D Series systems, the NeWS font manager provides compatibility with other window systems and programs as well. (For information on the NeWS font manager, refer to the *4Sight User's Guide, Volume I: Programming Guide.*) The font information in this chapter is provided for historical (as well as performance) reasons.

3.1 Characters

Use `cmov` and `charstr` to draw text. `cmov` determines where the system draws text on the screen, and `charstr` draws a string of characters.

The character string is drawn in the current font, which, by default, is a fixed-width sans-serif font 9 pixels wide. Strings drawn with raster fonts are not scaled; unless special care is taken, when a labeled object shrinks as it moves away from the viewer, for example, the label stays the same size. Similarly, no matter what rotation is in effect, the character string maintains the same orientation (horizontal for any standard font), because fonts are defined in 2-D, with respect to the raster display, and scaling, rotating, or translating such 2-D primitives has no meaning in a 3-D context.

cmov

The current character position determines where the system draws text on the screen. `cmov` moves the current character position to a specified point (as `move` sets the current line drawing position). *x*, *y*, and *z* are integers, shorts, or real numbers in 2D or 3D that specify a point in world coordinates. `cmov` transforms the world coordinates into window coordinates, which becomes the new character position. `cmov` does not affect the current graphics position.

The current character position is transformed in exactly the same way as a vertex. If, for example, it is clipped out by the current viewing transformation, the current character position is set to invalid, and any character strings that are drawn do not appear. `cmov` does not draw anything—it simply sets the character position where drawing will occur when `charstr` is issued.

	2-D	3-D
short integer	cmov2s	cmovs
long integer	cmov2i	cmovi
float	cmov2	cmov

Table 3-1. The cmov Subroutine

The argument lists are: `cmov2 (x, y)` and `cmov (x, y, z)`.

Note that character position includes *z* values as well as *x* and *y* values. Because of this, you can z-buffer and depthcue characters.

charstr

`charstr` draws a string of raster characters. The origin of the first character in the string is the current character position. After the system draws the string, it updates the current character position to the pixel to the right of the last character in the string. Character strings are null-terminated in C. The text string is drawn in the current font and color.

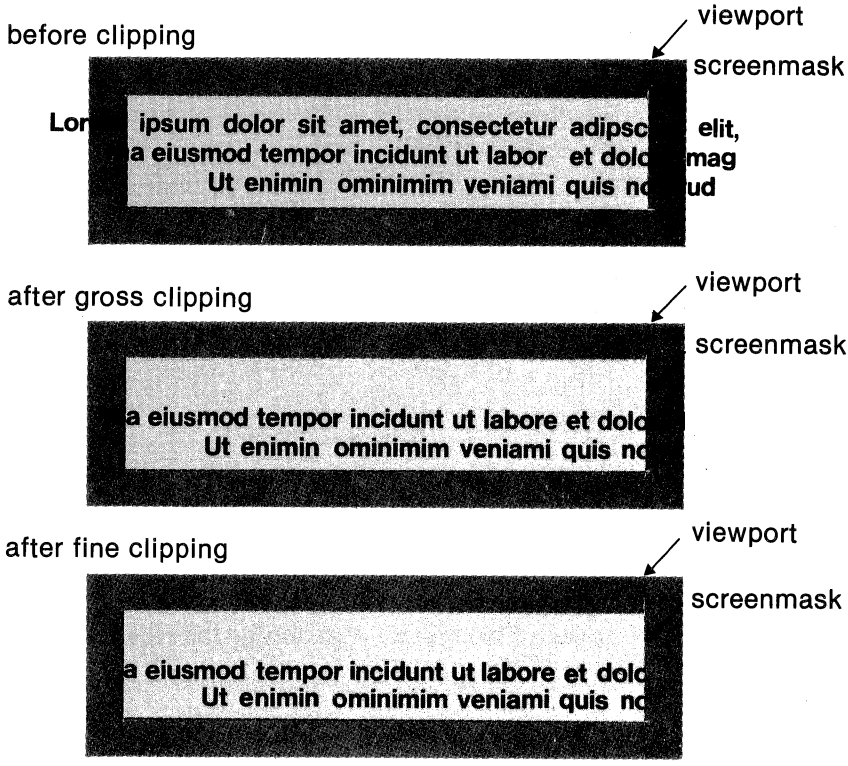
```
void charstr(str)
String str;
```

If the origin of a character string lies outside the viewport, no characters in the string are drawn. If the origin is inside the viewport, the characters are individually clipped to the screenmask. When the viewport is larger than the screenmask, character strings that begin inside the viewport are clipped to the screenmask. This process is called *fine clipping*. Character strings that begin outside the viewport are clipped out; this is called *gross clipping*. For an illustration of character clipping, see Figure 3-1.

getcpos

Similar to `getgpos` but for character position, `getcpos` returns the current character position's screen coordinates into the locations pointed to by *ix* and *iy*.

```
void getcpos(ix, iy)
Screencoord *ix, *iy;
```



Gross clipping removes all strings that start outside the viewport. Fine clipping trims individual characters to the screenmask.

Figure 3-1. Gross and Fine Clipping

The following example draws two lines of text. The program assumes that the default font is less than 12 pixels high.

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

main()
{
    prefposition(100, 500, 100, 500);
    winopen("rasterchars2");
    color(BLACK);
    clear();
    color(RED);
    cmov2i(50,80);
    charstr("The first line is drawn ");
    charstr("in two parts. ");
    cmov2i(50, 66);
    charstr("This line is 14 pixels lower. ");
    sleep(3);
    gexit();
    exit(0);
}
```

This program illustrates two things. First, notice that the first line is drawn in two parts. The first `cmov2i` sets the current character position to 50 pixels over and 80 pixels up from the lower-left corner of the window. After the first string is drawn, the current character position is advanced to follow the space character at the end of the line. When "`in two parts.`" is drawn, it continues from the current character position. Finally, the character position is reset to start below the beginning of the top line, and the second line is drawn.

Note that the characters are drawn in the current color (RED). Because nothing was mentioned in the program about fonts, all the strings are drawn in the current font (font 0), which is defined when `winopen` is called.

The next example uses `rotate` (a subroutine that has not yet been covered) that illustrates that character strings are drawn in the same orientation no matter where they move, and that `cmov` is transformed like any other geometry. In the example, `rotate` rotates about the *z* axis (coming directly out of the screen) by 5 degrees each time. The rotation is about the origin, so vertex *p1* should remain fixed. The vertex labels rotate with the vertices.

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

```
float p1[] = {0.0, 0.0};
float p2[] = {0.6, 0.0};
float p3[] = {0.0, 0.6};

main()
{
    long i;

    prefposition(100, 500, 100, 500);
    winopen("rasterchars");
    ortho2(-1.0, 1.0, -1.0, 1.0);
    for (i = 0; i < 40; i++) {
        color(BLACK);
        clear();
        rotate(50, 'z');
        color(RED);
        bgnpolygon();
            v2f(p1); v2f(p2); v2f(p3);
        endpolygon();
        color(GREEN);
        cmov2(0.0, 0.0);
        charstr("vert1");
        cmov2(0.6, 0.0);
        charstr("vert2");
        cmov2(0.0, 0.6);
        charstr("vert3");
        sleep(1);
        gexit();
        exit(0);
    }
}
```

strwidth

`strwidth` returns the width of a text string in pixels, using the character spacing parameters in the current raster font. The string can be any null-terminated ASCII string of characters. Note that all characters in some fonts might not be the same width, so `strwidth` does not necessarily return the width of a character times the number of characters in the string. The default font has a fixed width of nine pixels, so if that is the current font, `strwidth` returns nine times the string's length.

```
long strwidth(str)
string str;
```

3.2 Fonts

A font on the IRIS-4D is a collection of up to 255 rectangular arrays of masks. If a 1 appears in a mask, then the corresponding pixel is turned on to the current color and if a 0 appears, the pixel is left as it is. For example, the following bitmasks might be used to draw the character "A":

Binary		Hexadecimal
0000011000000000	=	0x0600
0000011000000000	=	0x0600
0000111100000000	=	0x0F00
0000111100000000	=	0x0F00
0001100110000000	=	0x1980
0001100110000000	=	0x1980
0011000011000000	=	0x30C0
0011111111000000	=	0x3FC0
0110000001100000	=	0x6060
0110000001100000	=	0x6060
1100000000110000	=	0xC030
1100000000110000	=	0xC030

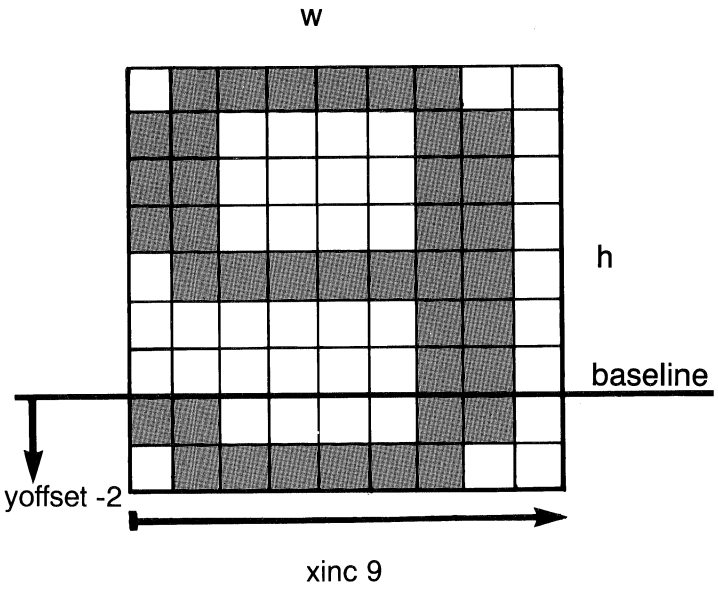
A font can have definitions for any character value between 1 and 255. Typically, the bitmask entry for each ASCII character is a mask that draws that character. For the example above, the ASCII value of "A" is 65 (decimal), so entry 65 in the font is associated with the bitmask above. If such a font were defined, the string "AAA" would draw three copies of the character whose bitmask appears above.

In addition to the bitmask information for each character, you need to know the width and height of the character in pixels. The width cannot be inferred from the bitmask, since all bitmask data comes in 16-bit words. In the letter *A* example above, the width is 12 bits—at most 12 bits are written horizontally for this character. The height above this character "A" is also 12 bits.

Normally, a character's origin is at the lower-left corner of the bitmask, as is the case for the *A* above. The character is placed by placing its bitmask so that the bitmask's origin is at the current character position. For a character with a descender, such as *g*, *j*, or *y*, you needthe lowest couple of lines of the bitmark to lie below the current character position, so the origin should not be at the lower-left corner. Two values, the *xoffset* and the *yoffset*, tell how far the character's origin must be moved to bring it to the lower-left corner. For characters with descenders, *yoffset* is typically negative (see Figure 3-2).

Finally, another number for each character indicates how far to the right the current character position must be advanced after drawing the character. This is usually different from the width, and is labeled the *x* increment. In the *A* example above, the character position would probably be advanced by 14 pixels to leave a little space between it and the next character.

To define a single character, you need the bitmask itself, and the width, height, *xoffset*, *yoffset*, and *xincrement*. The routine `defrasterfont` allows you to define a collection of characters so described (see Figure 3-2).




```
defrasterfont (n, ht, nc, chars, nr, rasters);

chars ['g']  = {   724,        8,    9,     0,      -2,     9  }
                   byte offset  w     h   xoffset  yoffset  xinc
                   into rasterarray
short
rasterarray [ ]      = { . . .
                         . . .
position 724 >             0x7E00,  0xC300,  0x0300,  0x0300,
                           0x7F00,  0xC300,  0xC300,  0xC300,
                           0x7E00,
                         . . .
                       }
```

Raster font characters are defined by a bitmap, 1 bit per pixel. The width and height of the character, the number of bits in one row of the bitmap, and the baseline position are also specified. See the manual page *defrasterfont* in the Reference Guide for a more complete example.

Figure 3-2. defrasterfont

To simplify matters, the character bitmasks are packed together in one array of 16-bit values (shorts), so the bitmask is determined by the offset into the bitmask array. For example, if the font contains the letter *A* above as its first character, and a bitmask for *B* as its second, the offset for *B* is 12 shorts (the length of the bitmask definition of *A*). Note that the length and width together determine the number of shorts in a character's definition.

defrasterfont

`defrasterfont` defines a raster font. It takes six arguments—n, ht, nc, chars, nr, and raster:

- *n* is a font number
- *ht* is an integer that specifies the maximum height of the font characters in pixels
- *nc* gives the number of characters in the font, which is the number of elements in the *chars* array
- *chars* contains a description of each character in the font. The description includes the height and width of the character in pixels, the offsets from the character origin to the lower-left corner of the bounding box, an offset into the array of rasters, and the amount to add to the current character *x* position after drawing the character. *chars* is an array of structures of type Fontchar, defined in the standard header file *gl/gl.h*. Figure 3-2 gives a sample character definition.
- *raster* is an array of *nr* shorts of bitmap information. It is a one-dimensional array of bitmask bytes ordered from left to right, then bottom to top. Mask bits are left-justified in the character's bounding box.

```
void defrasterfont(n, ht, nc, chars, nr, raster)
short n, ht, nc, nr;
Fontchar chars[];
short raster[];
```

Font 0 is the default raster font, which you cannot redefine. It is a Helvetica-like font with fixed-pitch characters. If the viewport is set to the whole screen, approximately 142 of the default characters fit on a line (1 character occupies 9 pixels). If baselines are 16 pixels apart, 64 lines fit on the screen.

font

`font` selects the font that the system uses whenever `charstr` draws a text string. Its argument is the font number assigned to the font built by `defrasterfont`. This font remains the current font until you use `font` to select another font.

```
font(fntnum)
short fntnum;
```

Here is a sample program that defines a font with three characters—a lowercase *j*, an arrow, and the Greek letter sigma. The *j* is assigned to the ASCII value of *j*, and the arrow and sigma are assigned to ASCII values 1 and 2 (written \001 and \002 in the C code). Two sample strings are then written out, the first of which contains only characters that are defined, while the second contains undefined characters. Note that when characters are not defined, no error occurs—but nothing is drawn out for them.

```
/* Define a font with three characters -- a lower-case j,
 * an arrow, and a greek sigma.  Use ascii values 1 and 2
 * (\001 and \002) for the arrow and sigma.  Use the
 * ascii value of j (= \152) for the j character.
 */

#include <gl/gl.h>

#define EXAMPLEFONT     1

#define efont_ht     16
#define efont_nc     127

unsigned short efont_bits[] = {
/* lower-case j */
0x7000, 0xd800, 0x8c00, 0x0c00, 0x0c00, 0x0c00, 0x0c00,
0x0c00, 0x0c00, 0x1c00, 0x0000, 0x0000, 0x0c00, 0x0c00,

/* arrow */
0x0200, 0x0300, 0x0380, 0xafc0, 0xafe0, 0xaff0, 0xafe0,
0xafc0, 0x0380, 0x0300, 0x0200,

/* sigma */
0xffc0, 0xc0c0, 0x6000, 0x3000, 0x1800, 0x0c00, 0x0600,
0x0c00, 0x1800, 0x3000, 0x6000, 0xc180, 0xff80,
```

```
};

#define efont_nr (sizeof efont_bits)

#define ASSIGN(fontch, of, wi, he, xof, yof, wid) \
                fontch.offset = of; \
                fontch.w = wi; \
                fontch.h = he; \
                fontch.xoff = xof; \
                fontch.yoff = yof; \
                fontch.width = wid

Fontchar efont_chars[127];

main () {

    ASSIGN(efont_chars['j'],     0,  6, 14, 0, -2,  8);
    ASSIGN(efont_chars['\001'], 14, 12, 11, 0,  0, 14);
    ASSIGN(efont_chars['\002'], 25, 10, 13, 0,  0, 12);

    winopen("font");
    color(BLACK);
    clear();
    defrasterfont (EXAMPLEFONT, efont_ht, efont_nc,
                    efont_chars,
                    efont_nr, efont_bits);
    font (EXAMPLEFONT);
    color(RED);
    cmov2i(100, 100);
    charstr ("j\001\002\001jj\002");
    cmov2i(100, 84);
    charstr("ajb\001c\002d");
    sleep(10);
    gexit();
    exit(0);

}
```

3.3 Font Query Subroutines

The following subroutines return information about the current font—what number it is, how high the characters are, and the maximum descender any character has.

getfont

`getfont` returns the index of the current raster font.

```
long getfont()
```

getheight

`getheight` returns the maximum height of a character in the current raster font, including ascenders (present in tall characters, such as the letters *t* and *h*) and descenders (present in such characters as the letters *y* and *p*, which descend below the baseline). It returns the height in pixels.

```
long getheight()
```

getdescender

`getdescender` returns the longest descender in the current font. It returns the number of pixels the longest descender goes below the baseline.

```
long getdescender():
```

4. Display and Color Modes

The framebuffer on the IRIS-4D is organized as a set of bitplanes. One bitplane contains exactly 1 bit of information for each pixel on the screen. The number of bitplanes varies from system to system. These bits store not only color information, but information about depth, overlays and underlays, and, on a GT with alpha planes, an alpha channel.

As described in Chapter 6, smooth motion requires the system to keep two copies of the color information. This chapter discusses only color information—for discussions of smooth motion (double buffering), refer to Chapter 6; for depth information (z-buffering), Chapter 8; for overlays and underlays, Chapter 11; and for the alpha buffer, Chapter 15.

The creation of graphics includes two basic steps: first, the drawing subroutines write data into the bitplanes, and second, the display hardware interprets that data as colors on the screen. The Graphics Library 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.

This chapter begins with a general discussion of color and how the settings of the three color guns relate to the colors displayed on the screen. It is followed by a discussion of RGBmode, a color mode that gives direct access to the settings of the color guns. Gouraud shading in RGBmode, which allows you to draw smoothly shaded figures, is then covered, followed by a discussion of color maps and how to draw shaded objects using a color map. Finally, a series of miscellaneous subroutines related to color are described, including gamma ramps, multimap mode, blinking colors, and getting the current color.

4.1 Color Display

If you have a standard monitor, it has three color guns that sweep out the entire screen area 60 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. Consider just the red color gun. If no electrons are fired at a pixel, its phosphors do not glow at all, and it appears black. 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 black = (0,0,0), (1,1,1), (2,2,2), ..., (255,255,255) = white are different shades of gray ranging from black to white. Since 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, *Fundamentals of Interactive Computer Graphics*, Addison-Wesley Publishing Co., 1982. Also, the first example in Section 4.2 allows you to experiment with the correspondence between RGB triples and perceived colors.

4.2 RGB mode

The simplest way to interpret pixel data is to provide 8 bits each (255 values) for red, green, and blue, and to display exactly those values of red, green, and blue on the screen. This is called RGB mode and, as described here, it requires 24 bits of data per pixel.

The color subroutines used in the previous examples take only a single argument, e.g., BLACK, GREEN, etc. This is the default, color map mode, which is discussed later in this chapter.

The following sample program draws 64 rectangles in RGB mode and illustrates a few of the possible colors available.

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

short whitevec[3] = {255, 255, 255};
short blackvec[3] = {0, 0, 0};

main()
{
    long red, green, blue;
    long i, j, k, majorx, majory;
    char string[100];
    short rgbvector[3];

    prefposition(0, 1023, 0, 1023);
    winopen("rgbdemo");
    RGBmode();
    gconfig();
    c3s(blackvec);
    clear();
    for (i = 0; i < 4; i++) {
        rgbvector[0] = red = i*85;
        for (j = 0; j < 4; j++) {
            rgbvector[1] = green = j*85;
            for (k = 0; k < 4; k++) {
                rgbvector[2] = blue = k*85;
                c3s(rgbvector);
                if (i < 2)
                    majory = 80;
                else
                    majory = 520;
```

```
                if (i == 0 || i == 2)
                    majorx = 80;
                else
                    majorx = 520;
                rectfi(majorx + 110*j, majory + 110*k,
                    majorx + 110*j + 80, majory + 110*k + 80);
                c3s(whitevec);
                cmov2i(majorx + 110*j, majory + 110*k - 12);
                sprintf(string,"%d %d %d",red,green,blue);
                charstr(string);
            }
        }
    }
    sleep(3);
    gexit();
    exit(0);
}
```

This program draws 64 squares with all combinations of red, green, and blue chosen from 0, 85, 170, and 255. Each square is labeled with its RGB triple below it.

After opening the window in the usual way, `RGBmode` is called, followed by `gconfig`. `RGBmode` is one of many subroutines that must be followed by `gconfig` before it takes effect. Machine reconfiguration is not a simple task, so the most efficient way to do it is to give a series of subroutines that describe the new configuration, and then call `gconfig` to reconfigure once. Calls to `gconfig` are not automatically performed after each subroutine that requires reconfiguration, because too many calls to it in one piece of code might hurt performance. This simple example changes only one part of the configuration (to RGB mode) and you must call `gconfig` to make it take effect.

The next subroutine, `c3s`, sets the color to the RGB triple specified by the vector *blackvec*, which is initialized to contain zeros for the red, green, and blue components. Like the vertex subroutines described in Chapter 2, the color subroutines have various sizes (three-component and four-component) and types: short integers (16 bits), long integers (32 bits), or floating point values. 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 (or 1.0).

Table 4–1 lists the subroutines that set the RGB color.

	RGB	RGBA
16-bit integer	c3s	c4s
32-bit integer	c3i	c4i
32-bit floating point	c3f	c4f

Table 4–1. The c Subroutine

The floating point versions are different because the floating point range is not 0 to 255, but rather 0.0 to 1.0. This is useful for lighting calculations (Chapter 9). The floating point range from 0.0 to 1.0 is mapped linearly to the range from 0 to 255. Floating point values larger than 1.0 are mapped to 255.

The rest of the code is straightforward—the color vector is built up to contain the requisite red, green, and blue components, and 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 the red, green, and blue components.

After drawing each rectangle, the color is changed to white (255,255,255), and the string describing the red, green, and blue components of the color are printed underneath. If you have never worked with the red, green, and blue components of color, you might want to modify this program (or create another one) to allow you to experiment with different combinations.

Another convenient subroutine that specifies the red, green, and blue components of color is `cpack`. It 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. `cpack(0x01020304)` sets the red, green, blue, and alpha components to 4, 3, 2, and 1, respectively. Use `cpack` in exactly the same way you use the other color subroutines.

4.2.1 RGBcolor

For compatibility, the Graphics Library also contains an old-style subroutine to set the RGB color. It is `RGBcolor` and it takes as arguments the three individual components of the color.

```
void RGBcolor(red, green, blue)
short red, green, blue;
```

`RGBcolor` is supported for compatibility only; it is recommended that you use the `c3i`, `c3s`, `c3f`, or `cpack` subroutine in code for the IRIS-4D.

The following program illustrates the use of `RGBcolor` to clear the screen to the color (0,100,200):

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

main()
{
    prefposition(100, 500, 100, 500);
    winopen("RGBcolordemo");
    RGBmode();
    gconfig();
    RGBcolor(0, 100, 200);
    clear();
    sleep(3);
    gexit();
    exit(0);
}
```

4.3 Gouraud Shading

In the examples 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 geometric object you are modeling has only flat (polygonal) faces, this might be the way it should look; however, in many cases, the 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. You can 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.

The graphics hardware on the IRIS-4D/GT and Personal IRIS shades the polygons rapidly. (You can also shade polygons using lighting models, positions and colors of lights, surface properties, etc. See 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.

To achieve polygonal shading, the system linearly interpolates the (RGB) colors at each vertex along the edges connecting them, then interpolates the interpolated colors on the edges across the interior of the polygon. The result is a smooth color variation across the entire polygon.

To achieve Gouraud shading in 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. For example, suppose the edge of a polygon that is 6 pixels long is colored (0,20,100) at one end and (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). Notice that each of the color components changes smoothly from each 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 find the colors of the pixels on the interior. Figure 4–1 shows the result of shading a triangle whose vertices have colors (0,20,100), (75,60,50), and (0,0,0).

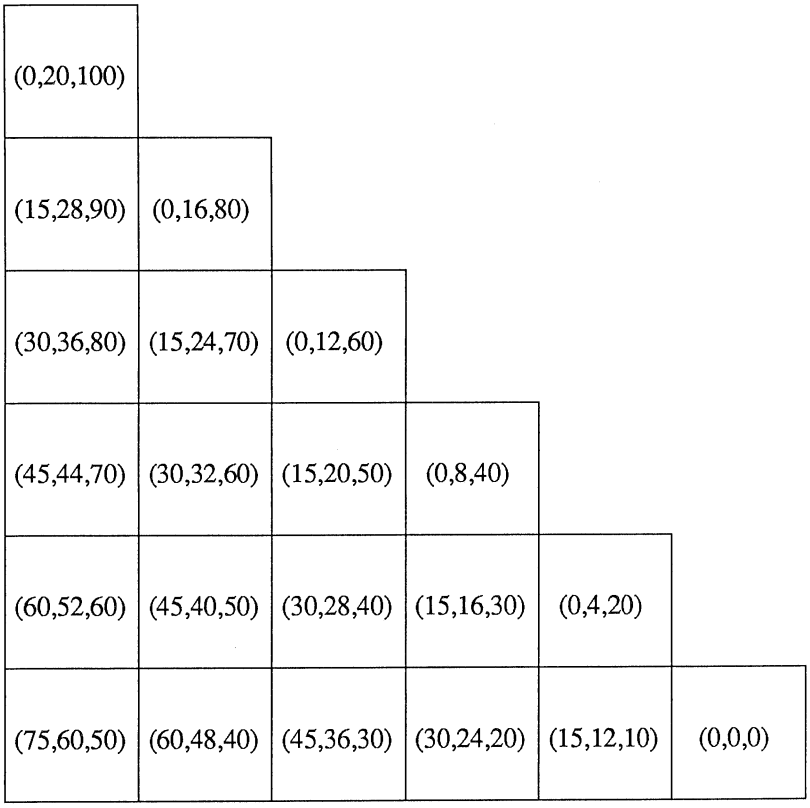


Figure 4–1. Shaded Triangle

To shade a polygon, set the color before each vertex. The following program draws a large 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 on
this machine\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, 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.

To shade a polyline, do the same thing—set the color before each vertex between `bgnline` and `endline`.

Typically, polygons are small and do not have wildly different colors at their vertices as the example above does. The following program implements a simple lighting model in user code and uses it to draw a cylinder. Real programs would not implement simple lighting calculations in user code; they would take advantage of the high-performance lighting hardware in the graphics pipeline.

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

#define RADIUS .9

#define max(x,y) (((x) > (y)) ? (x) : (y))

float whitevec[3] = {1.0, 1.0, 1.0};
float blackvec[3] = {0.0, 0.0, 0.0};
float light[3] = {3.0, 0.0, 1.2};
float dot(), dist2();

main()
{
    float p0[3], p1[3], p2[3], p3[3];
    float c0[3], c1[3], c2[3], c3[3];
    float n0[3], n1[3], n2[3], n3[3];
    float theta, x, dx, dtheta;
    long n;

    prefposition(100, 600, 100, 600);
    winopen("cylinder");
    RGBmode();
    gconfig();
    ortho2(-1.5, 1.5, -1.5, 1.5);
    c3f(blackvec);
    clear();
    for (n = 1; n < 10; n++) {
        dx = 3.0/n;
```

```
        dtheta = M_PI/n;
    for (x = -1.5; x < 1.5; x = x+dx) {
        for (theta = 0.0; theta < M_PI; theta += dtheta) {
        p0[0] = p1[0] = x;
        p0[1] = p3[1] = RADIUS*cos(theta);
        p0[2] = p3[2] = RADIUS*sin(theta);
        p1[1] = p2[1] = RADIUS*cos(theta+dtheta);
        p1[2] = p2[2] = RADIUS*sin(theta+dtheta);
        p2[0] = p3[0] = x+dx;
        n0[0] = 0; n0[1] = p0[1]/RADIUS; n0[2] =
                        p0[2]/RADIUS;
        n1[0] = 0; n1[1] = p1[1]/RADIUS; n1[2] =
                    p1[2]/RADIUS;
        n2[0] = 0; n2[1] = p2[1]/RADIUS; n2[2] =
                    p2[2]/RADIUS;
        n3[0] = 0; n3[1] = p3[1]/RADIUS; n3[2] =
                    p3[2]/RADIUS;
        c0[0]=c0[1]=c0[2]=max(0.0,dot(light,n0)/
                    (.5+dist2(light,p0)));
        c1[0]=c1[1]=c1[2]=max(0.0,dot(light,n1)/
                (.5+dist2(light,p1)));
        c2[0]=c2[1]=c2[2]=max(0.0,dot(light,n2)/
                (.5+dist2(light,p2)));
        c3[0]=c3[1]=c3[2]=max(0.0,dot(light,n3)/
                (.5+dist2(light,p3)));
        bgnpolygon();
            c3f(c0); v3f(p0);
            c3f(c1); v3f(p1);
            c3f(c2); v3f(p2);
            c3f(c3); v3f(p3);
        endpolygon();
        }
    }
    sleep(5);
    }
}

/* dot: find the dot product of two vectors */

float dot(v1, v2)
float v1[3], v2[3];
{
    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[3], v2[3];
{
    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***n* rectangles whose vertices lie on the surface of the cylinder. The equation for the cylinder is: $y^2 + z^2 = \text{RADIUS}^2$, where *y* and are parameterized by θ y = cos(θ), z = sin(θ), $0 \leq \theta \leq \pi$, 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, RADIUS * cos (θ), RADIUS * sin (θ)). At that point, the normal vector is: (0, cos (θ), sin (θ)).

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 only colors between white and black, i.e., 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 1 polygon, then 4, 9, 16, 25, ... 100 polygons. Each is presented for 5 seconds before the next one is drawn. The first one is completely black, since 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. Notice that there are a lot of artifacts in the first few images.

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.

In the code itself, *p0*, *p1*, *p2*, and *p3* are the points around each of the approximating rectangles,;*n0*, ..., *n3* are the corresponding normal vectors, and *c0*, ..., *c4* are the colors calculated at the points.

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*. Many applications are better suited to color map mode than to RGB mode, and it is currently the only color mode supported in the overlay, underlay, and popup bitplanes.

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 8 values in the color map are loaded as follows:

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 4–2. Lowest Eight Values of Color Map

In the examples in Chapters 1, 2 and 3, the subroutine 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. Since the display is in color map mode, pixel values of 2 were looked up in the color map, and the RGB triple (0,255,0) = bright green was presented.

These map entries or any other map entries can be changed. Since there is only one color map in the system, whenever you modify the map, it is modified for all the other applications running in different windows. However, applications running in RGB mode are not affected.

To enter color map mode, call `cmode` followed by `gconfig`. (The system is in color map mode by default; consequently, you need only call `cmode` if the system is in RGB mode.) `color` sets the current color to be used in drawing. It can be any number between 0 and the system's limit: 255 for the Personal IRIS in single-buffer mode, 15 for the Personal IRIS in double-buffer mode, and 4095 for other systems. Multimap mode, described in Section 4.4, is a special case. (Other limits apply to the overlay, underlay, and popup bitplanes.)

The only remaining question is how to change color map entries. This is done with `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. (Multimap mode has only 256 map entries. See Section 4.4, "Onemap and Multimap Modes.") The calling sequence is:

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

Note: On the IRIS-4D/G Series systems, the top 256 entries of the color map cannot be used if the system simultaneously has windows running in RGB and color map mode. This restriction does not exist on other IRIS-4D Series systems, but if you are writing code to run on all types of machines, do not use entries 3840 to 4095.

Because of their hardware architecture, IRIS-4D/G Series systems cannot simultaneously run double-buffered windows in RGB mode and color map mode.

The following sample program 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 code uses the predefined BLACK to clear the screen).

```
#include <gl/gl.h>
#define GRAY 8
#define PINK 9
main()
{
    prefposition(100, 500, 100, 500);
    winopen("pink-n-gray");
    mapcolor(GRAY, 150, 150, 150);
    mapcolor(PINK, 255, 80, 80);
    color (BLACK);
    clear();
    color (GRAY);
    recti(0, 0, 100, 100);
    color(PINK);
    circi(50, 50, 50);
    sleep(3);
    gexit();
    exit(0);
}
```

color and colorf

In the second program in Section 4.2.4, the `color` and `colorf` statements set the color index in the current draw mode of the active framebuffer: normal, pop-up, overlay, or underlay. 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. The color indices of all framebuffers are set to 0 when `gconfig` is called.

`colorf` is identical to `color`, except that it expects a floating point value. Before the value is written into memory, however, it is rounded to the nearest integer value. If you are using Gouraud shading, systems that iterate color indices with fractional precision yield more precise shading results with `colorf` than with `color`. IRIS-4D/B/G/GT/GTX systems and Personal IRIS systems with early serial numbers do not iterate with fractional precision,. To determine whether your system supports fractional iteration, use `getgdesc(GD_CIFRACT)`. The results of `color` and `colorf` are indistinguishable when using `shademodel(FLAT)`.

Do not call `color` or `colorf` when the framebuffer is in RGB mode.

4.4.1 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. Although the line produced might be pretty or interesting, it is probably not what you had in mind.

To shade in color map mode, you must first load a portion of the color map with a color ramp. Since the variation is one-dimensional (only the color index varies; in RGB mode, the red, green, and blue components can vary individually), the effects are more restricted.

The following sample program draws a color map mode, Gouraud-shaded polygon:

```
/***
rampshade.c - draw a colorindex gouraud-shaded polygon
***/

#include <gl/gl.h>

float *
pack3f(x, y, z)
float x, y, z;
{
static float v[3];
    v[0] = x;
    v[1] = y;
    v[2] = z;
    return v;
}

int
main()
{
int i;

    prefsize(300, 300);
    (void) winopen("shade");
    color(BLACK);
    clear();

    /* create a magenta ramp but avoid first 16 colors */
    for (i = 0;  i < 256;  i++)  mapcolor(i + 16, i, 0, i);

    bgnpolygon();
    color(0 + 16);
    v3f(pack3f(50.0, 50.0, 0.0));
    color(100 + 16);
    v3f(pack3f(200.0, 50.0, 0.0));
    color(250 + 16);
    v3f(pack3f(250.0, 250.0, 0.0));
    color(100 + 16);
    v3f(pack3f(50.0, 200.0, 0.0));
    endpolygon();
```

```
    sleep(3);
    gexit();
    exit(0);
}
```

The Personal IRIS supports dithered rendering, and uses it to approximate the color represented by non-integral color indices. The following sample program is a version of the previous one that interpolates colors between the adjacent color indices 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, "Fractional color index shading 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;
}
```

4.4.2 Blinking

(Blinking is an advanced topic and 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. *rate* indicates the number of vertical retraces at which the system updates the color located at *i* in the current color map. (See Section 4.4, "Onemap and Multimap Modes," for a definition of vertical retrace.) Each *rate* retraces, the color map entry *i* is remapped so that it alternates between the new *red, green, blue* value and the original value. You can have 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 when *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.

```
blink(rate, i, red, green, blue)
short rate;
Colorindex i;
short red, green, blue;
```

The following program draws 20 rectangles on the window, and blinks each one between white and red at a different rate. The leftmost rectangle blinks every retrace, the second blinks every second retrace, the third blinks every third retrace, and so on. After the blinking rectangles are displayed for 20 seconds, `blink` is called with a rate of -1 to turn off all the blinking in the system. If you kill this program before it completes, map entries 110 through 119 continue to blink.

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

main()
{
    register i, j;
    prefposition(100, 800, 100, 800);
    winopen("blinker");
    ortho2(190.0, 600.0, 90.0, 510.0);
    color(BLACK);
    clear();
    for (i = 10; i < 30; i++) {
        mapcolor(i+100, 255, 255, 255);
        color(i+100);
        rectfi(i*20, 100, i*20 + 10, 500);
        blink(i-9, i+100, 255, 0, 0);
    }
    sleep(20);
    blink(-1, 0, 0, 0, 0);
    gexit();
    exit(0);
}
```

4.5 Getting Color Information

The Graphics Library provides three subroutines to get color information: `getcolor`, `gRGBcolor` and `getmcolor`. In most cases, `getcolor` and `RGBcolor` simply return the most recently set color or RGB color triple; however, when using the automatic lighting models, the system can change the current color as a side effect of lighting calculations. `getmcolor` returns the setting of the color map for a given index.

getcolor

`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.

```
long getcolor()
```

gRGBcolor

`gRGBcolor` returns the current RGB values. Use `gRGBcolor` only in RGB mode.

```
void gRGBcolor(red, green, blue)
short *red, *green, *blue;
```

getmcolor

`getmcolor` returns the red, green, and blue components of a color map entry. *i* is the index into the color map, and *red*, *green*, and *blue* are the RGB intensities associated with that index.

```
void getmcolor(i, red, green, blue)
Colorindex i;
short *red, *green, *blue;
```

4.6 Onemap and Multimap Modes

(This is an advanced topic; you can skip it on the first reading.) `onemap` organizes the color map as a single map with 4096 entries (256 on some systems). `multimap` organizes the color map as 16 small maps, each with a maximum of 256 RGB entries.

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 mode is useful on systems with only 8 bitplanes (or 16 in double-buffer mode; see Section 6.2) but it can be used on other systems for techniques such as color map animation.

Call `gconfig` to activate the `onemap` or `multimap` settings.

multimap

`multimap` organizes the color map as 16 small maps, each with a maximum of 256 RGB entries. The number of entries in each map is 256. `multimap` does not take effect until `gconfig` is called.

```
void multimap()
```

onemap

`onemap` organizes the color map as a single map with 4096 entries (256 on some systems). You must call `gconfig` for `onemap` to take effect. `onemap` is the default mode.

```
void onemap()
```

You can use the following routines with `onemap` and `multimap`: `getcmmode` , which returns the current color map mode; `setmap`, which selects which of the small maps the system uses in multimap mode; `getmap`, which returns the number of the current color map; and `cyclemap`, which cycles through the color maps at a selected rate.

getcmmode

`getcmmode` returns the current color map mode. FALSE indicates multimap mode; TRUE indicates onemap mode.

```
Boolean getcmmode()
```

setmap

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. `setmap` selects which of the small maps (0 through 15) the system uses in multimap mode.

```
void setmap(mapnum)
short mapnum;
```

getmap

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

```
long getmap()
```

cyclemap

`cyclemap` cycles through color maps at a specified rate. It defines a duration (in vertical retraces), the current map, and the map that follows when the duration lapses. (See Section 4.3, "Blinking," for a definition of vertical retrace.) For example, the following routines set up multimap mode and cycle between two maps, leaving map 1 on for ten vertical retraces and map 3 on for five retraces.

```
multimap();
gconfig();
cyclemap(10, 1, 3);
cyclemap(5, 3, 1);
```

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

```
void cyclemap(duration, map, nextmap)
short duration, map, nextmap;
```

4.8 Gamma Correction

(This is an advanced topic; you can skip it 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, however, is not 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 nonlinear response.

gammaramp

`gammaramp` supplies another level of indirection for all color map and RGB values. It affects only the display of color, not the values that are written in the bitplanes. You use it to provide gamma correction, to equalize monitors with different color characteristics, or to modify the color warmth of the monitor.

`gammaramp` affects the entire screen and all running processes. It stays in effect until another call to `gammaramp` is made, or until the graphics hardware is reset. The default setting assigns the gamma value a curve with a slope of 1.7.

When objects are drawn on the screen in RGB mode, red, green, and blue are stored in the bitplanes and are displayed as (*r*[red], *g*[green], *b*[blue]) where *r*,*g*,*b* are the arrays last specified by `gammaramp`. Similarly, in color map mode if color *i* is mapped to red, green, blue, objects written in color *i* are displayed as (*r*[red], *g*[green], *b* [blue]).

```
void gammaramp(r,g,b)
short r[256],g[256],b[256]
```

Usually, the gamma ramp map is loaded with gamma corrections, but it can be loaded with any values. Note that on IRIS-4D/G systems, the gamma ramp is stored in the top 256 entries of the color map.

The following program takes a floating point argument, *gamma*, calculates a standard gamma ramp, and installs it using `gammaramp`.

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

short tab[256];

main(argc,argv)
int argc;
char **argv;
{
    int i;
    float gamma;

    if(argc<2) {
        fprintf(stderr, "usage: %s value\n", argv[0]);
        exit(1);
    }
    gamma = atof(argv[1]);
    noport();
    winopen("setgamma");
    for(i=0; i<256; i++)
        tab[i] = 255.0*pow(i/255.0,1.0/gamma)+0.5;
    gammaramp(tab,tab,tab);
    gexit();
    exit(0);
}
```

The program sets the same gamma ramp for red, green and blue, although a more general mapping is possible. In addition, the 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. You can open a tiny window, but nothing of interest is displayed in it. (See the *4Sight Programmer's Guide* for more information about `noport`.)

As a final example of the gamma ramp, here is a program that 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 intenstiy, and the lowest third to lowest intensity. It basically provides thresholding in red, green, and blue simultaneously. 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 <gl/gl.h>

main(argc,argv)
int   argc;
char *argv[];
{
    nt nsteps;

    noport();
    winopen("stepmap");
    if (argc != 2) {
        fprintf(stderr, "usage: %s numsteps\\n", argv[0]);
        exit(1);
    }
    nsteps = atoi(argv[1]);
    if (nsteps >= 2)
        stepgen(nsteps);
    gexit();
    exit(0);
}

short ramp[256];

stepgen(nsteps)
int nsteps;
{
    int i, val;
    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);
}
```

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

5. Input Subroutines

The Graphics Library supports three classes of input devices:

- *valuators*, which return an integer value. For example, a dial on a dial and button box is a valuator. A mouse is a pair of valuators: one reports horizontal position and the other reports vertical position.

- *buttons*, which return a Boolean value: FALSE when they are not pressed (open) and TRUE when they are pressed (closed). Keys on an unencoded keyboard, buttons on a mouse, and the switches on a dial and button box are all buttons.

- *pseudo-devices*, which return information about other system events. For example, the keyboard returns ASCII characters. Most of these pseudo-devices register events. The keyboard device reports character values when keys (or combinations of keys) are pressed. If you press the `a` key, an ASCII `a` is reported; if you press the `<shift>` key, nothing is reported, but if you hold down the `<shift>` key and then press the `a` key, an ASCII `A` is reported.

Devices are named by a unique integer within the domain of 1 to 32767, inclusive. Ranges within this domain are set aside for the three device classes. Table 5-1 shows how the domain is organized.

	Range	Device Class
	0x001 — 0x0FF	Buttons
1-0xFFF:	0x100 — 0x1FF	Valuators
Reserved	0x200 — 0x2FF	Pseudo-devices
Devices	0x300 — 0xEFF	Reserved
	0xF00 — 0xFFF	Additional Buttons
0x1000 — 0x7FFF:	0x1000 — 0x2FFF	Buttons
User-definable	0x3000 — 0x3FFF	Valuators
Devices	0x4000 — 0x7FFF	Pseudo-devices

Table 5-1. Class Ranges in the Device Domain

In addition to input subroutines, there are subroutines that control the characteristics of the peripheral input/output devices. These subroutines turn the keyboard click on and off, ring the keyboard bell, and control the lights and text on the dial and button box. See Tables 5-2 and 5-3 for a listing of devices and their descriptions. Section 5.4 describes other special devices.

Devices	Description
MOUSEX	x valuator on mouse
MOUSEY	y valuator on mouse
DIAL0...DIAL7	Dials on dial and button box
BPADX	x valuator on digitizer tablet
BPADY	y valuator on digitizer tablet
CURSORX	x valuator attached to cursor (usually MOUSEX)
CURSORY	y valuator attached to cursor (usually MOUSEY)
GHOSTX	x ghost valuator
GHOSTY	y ghost valuator
TIMER0...TIMER3	Timer devices

Table 5-2. Input Valuators

Devices	Description
MOUSE1	Right mouse button
MOUSE2	Middle mouse button
MOUSE3	Left mouse button
RIGHTMOUSE	Right mouse button
MIDDLEMOUSE	Middle mouse button
LEFTMOUSE	Left mouse button
SW0...SW31	32 buttons on dial and button box
AKEY...PADENTER	All the keys on the keyboard
BPAD0	Pen stylus or button for digitizer tablet
BPAD1	Button for digitizer tablet
BPAD2	Button for digitizer tablet
BPAD3	Button for digitizer tablet
MENUBUTTON	Menu button

Table 5-3. Input Buttons

Facilities exist to let you define your own input devices. Additional information is available in the *Graphics Library Reference Manual, C Edition.*

5.1 Polling and Queueing

Buttons and valuators have an associated value. If the input device is a button, the value is either 1 or 0. If the device is a valuator, such as a dial or the *x* position of the mouse, its value is an integer that indicates the position of the device. Pseudo-devices do not have a current value. It does not make much sense to talk about the current ASCII character on the keyboard.

A program can get the value from input devices in two ways: polling or queueing.

- Polling immediately returns the value of a device that is a button or valuator. For example, `getbutton(LEFTMOUSE)` returns 1 if the left button of the mouse is down and returns 0 if it is up.

- Queueing uses an event queue to save changes in device values and other input events so the program can read them later.

In most programs, you queue buttons so that the program does not miss a rapid up-and-down transition of the button. You can queue or poll valuators depending on the situation. If a program can keep up with valuator changes, then it is usually more efficient to queue the valuator because the queue captures all changes. Polling, however, always returns the most current value and can often give a more responsive feel to the user.

Another difference between queuing and polling is the effect that the window system has on them. When you queue buttons and valuators, an event is generated only for the process controlling the window that currently has focus. Polling, on the other hand, returns the current value of a device regardless of which window has focus. Programs that use polling should watch for changes to input focus and adjust their behavior accordingly.

You can decide which devices, if any, to queue, and establish rules about what constitutes a state change, or *event*, for those devices. By default, only the pseudo-devices `INPUTCHANGE` and `REDRAW` are queued.

5.2 Polling a Device

You can poll a device at any time to determine its current state.

getvaluator

You determine the values of valuators with `getvaluator`. You can poll any valuator, whether it is queued or not. The argument to `getvaluator` is a valuator device number (*val*). Its value reflects the current state of the device.

```
long getvaluator(val)
Device val;
```

getbutton

`getbutton` polls a button and returns its current state. *num* is the number of the device you want to poll. `getbutton` returns either TRUE or FALSE. TRUE indicates the button is pressed. This routine functions whether the device in question is queued or not.

```
Boolean getbutton(num)
Device num;
```

getdev

`getdev` polls up to 128 valuators and buttons at one time. You specify the number of devices you want to poll (*n*) and an array of device numbers (*devs*). (See Tables 5-1, 5-2, and 5-3 for listings of device numbers.) *vals* returns the state of each device in the corresponding array location.

```
void getdev(n, devs, vals)
long n;
Device *devs;
short *vals;
```

5.3 The Event Queue

Input devices can make entries in the event queue. Each entry includes the device number and a device value. `qdevice` enables queueing of events from an input device. `unqdevice` disables queuing of a device. `isqueued` tells you if a specific device is queued. The three subroutines most commonly used for queuing are `qdevice`, `qread`, and `qtest`.

The input queue can contain up to 101 events at a time. To check for overflow, you can queue the device `QFULL`. This inserts a `QFULL` event in the graphics input queue of a GL program at the point where queue overflow occurred. This event is returned by `qread` at the point in the input queue at which data was lost.

qdevice

`qdevice` queues the specified device (*dev*) (for example, a keyboard, button, or valuator). The argument of `qdevice` is a device number. Each time the device changes state, an entry is made in the event queue.

```
void qdevice(dev)
Device dev;
```

qtest

`qtest` returns the device number of the first entry in the event queue; if the queue is empty, it returns zero. `qtest` always returns immediately to the caller, and makes no changes to the queue.

```
long qtest()
```

qread

`qread`, like `qtest`, returns the device number of the first entry in the event queue. However, if the queue is empty, it waits until an event is added to the queue. `qread` returns the device number, writes the data part of the entry into the short pointed to by *data*, and removes the entry from the queue.

```
long qread(data)
short *data;
```

qreset

`qreset` removes all entries from the queue and discards them.

```
void qreset()
```

qgetfd

`qgetfd` allows a GL program to use the `select` system call to determine when there are events waiting to be read in the graphics input queue. A call to `qgetfd` returns a file descriptor that may be used as part of the `readfds` parameter of the `select` system call. When `select` indicates that the file descriptor associated with the graphics input queue is ready for reading, a call to `qread` or `blkqread` will not cause the program to block. See the description of `select` in the *IRIX Programmer's Reference Manual* for more information.

Example—Polling and Queueing

The following sample program uses both polling and queueing to control a simple drawing program.

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

#define X 0
#define Y 1
#define XY    2

main()
{
    short mval[XY], lastval[XY];
    Device mdev[XY];
    short val;
    long org[XY], size[XY];
    Boolean run;

    prefsize(400, 400);
    winopen("input");
    qdevice(LEFTMOUSE);
    qdevice(ESCKEY);
    color(BLACK);
    clear();
    getorigin(&org[X], &org[Y]);
    getsize(&size[X], &size[Y]);
    mdev[X] = MOUSEX;
    mdev[Y] = MOUSEY;
```

```
    run = TRUE;
    while (run) {
   switch (qread(&val)) {
   case LEFTMOUSE:
       if (val == 1) {
      getdev(XY, mdev, mval);
      color(WHITE);
      lastval[X] = mval[X] - org[X];
      lastval[Y] = mval[Y] - org[Y];
      while (getbutton(LEFTMOUSE) == TRUE) {
          getdev(XY, mdev, mval);
          mval[X] -= org[X];
          mval[Y] -= org[Y];
          if (mval[X] < 0 || mval[X] >= size[X] ||
         mval[Y] < 0 || mval[Y] >= size[Y])
         continue;
          else {
         bgnline();
             v2s(lastval);
             v2s(mval);
         endline();
         lastval[X] = mval[X];
         lastval[Y] = mval[Y];
          }
      }
       }
       break;
   case ESCKEY:
       if (val == 0)
      run = FALSE;
       break;
   }
    }
    gexit();
    return 0;
}
```

tie

You can tie a queued button to one or two valuators so that whenever the button changes state, the system records the button change and the current valuator position in the event queue. `tie` takes three arguments: a button *b* and two valuators *v1* and *v2*. Whenever the button changes state, three entries are made in the queue that record the current state of the button and the current position of each valuator. You can tie one valuator to a button by making *v2* equal to 0. You can untie a button from valuators by making both *v1* and *v2* equal to 0.

```
void tie(b, v1, v2)
Device b, v1, v2;
```

attachcursor

`attachcursor` attaches the cursor to the movement of two valuators. Both of its arguments, *vx* and *vy*, are valuator device numbers that correspond to the device that controls the horizontal and vertical location of the cursor, respectively. By default, *vx* is MOUSEX and *vy* is MOUSEY.

The valuators at *vx* and *vy* determine the cursor position in screen coordinates. Every time the values at *vx* or *vy* change, the system redraws the cursor at the new coordinates.

```
void attachcursor(vx, vy)
Device vx, vy;
```

To control cursor position from within a program, attach the cursor to `GHOSTX` and `GHOSTY`. The program can then use `setvaluator` on `GHOSTX` and `GHOSTY` to move the cursor.

curson and cursoff

`curson` and `cursoff` determine the visibility of the cursor in the current window. Use them to control the state of the cursor while drawing in the selected window.

`curson` makes the cursor visible in the current window; `cursoff` makes it invisible. By default, the cursor is on when a window is created.

Use `getcursor` to find out whether the cursor is visible. See Chapter 11, "Drawing Modes," for more information on `getcursor`.

```
void curson()
void cursoff()
```

noise

Some valuators are noisy; that is, they report small fluctuations in value, indicating movement when no event has occurred. `noise` allows you to set a lower limit on what constitutes a move. The value of a noisy valuator *v* must change by at least *delta* before the motion is significant. `noise` determines how often queued valuators make entries in the event queue. For example, `noise(v,5)` means that valuator *v* must move at least five units before a new queue entry is made.

```
void noise(v, delta)
Device v;
short delta;
```

qenter

`qenter` creates event queue entries. It places entries directly into the program's own event queue. `qenter` takes two 16-bit integers, *qtype* and *val*, and enters them into the event queue.

```
void qenter(qtype, val)
short qtype, val;
```

unqdevice

Use `unqdevice` to disable the queueing of events from a device. If the device has recorded events in the queue that have not been read, those events remain in the queue. (You can use `qreset` to flush the event queue.)

```
void unqdevice(dev)
Device dev;
```

isqueued

`isqueued` indicates whether a specific device is queued. It returns a Boolean value. TRUE indicates that the device is enabled for queuing; FALSE indicates that the device is not queued.

```
boolean isqueued(dev)
short dev;
```

blkqread

`blkqread` returns multiple queue entries. Its first argument, *data*, is an array of short integers, and its second argument, *n*, is the size of the array data. `blkqread` returns the number of shorts returned in the array *data*, which is filled alternately with device numbers and device values. Note that the number of entries read is twice the number of queue entries, hence it can be at most *n*.

You can also use `blkqread` when only the last entry in the event queue is of interest (for example, when a user-defined cursor is being dragged across the screen and only its final position is of interest).

```
long blkqread(data, n)
short *data
short n;
```

5.4 Special Devices

There are two types of special devices: keyboard and window devices.

5.4.1 Keyboard Devices

The keyboard device returns ASCII values that correspond to the keys typed on the keyboard. The device interprets keyboard movements in the standard manner; for instance, it reports an event only on a downstroke, taking into account the **<ctrl>** and **<shift>** keys.

Be careful to understand the difference between the device and the values it returns when you queue the keyboard. If your program contains the instruction:

```
qdevice(KEYBD);
```

then the statement:

```
dev = qread(&val);
```

returns the following:

```
dev = KEYBD
val = the ASCII integer index of the character pressed.
```

To test for individual keystrokes, you can use instructions of the format:

```
qdevice(AKEY);
```

This returns the device AKEY when the **A** key is pressed.

5.4.2 Window Manager Devices

The devices listed below are associated with window manager events. See Section 1, Using the GL Interface, in the *4Sight Programmer's Guide, Volume I.*

REDRAW	The window manager inserts a redraw token each time the window needs to be redrawn. The REDRAW token is queued automatically. The value returned is the window identification number of the window that the event pertains to.
REDRAWICONIC	Queues automatically when `iconsize` is called. The window manager sends this token when a window needs to be redrawn as an icon by the program itself. The value returned is the window identification number of the window that the event pertains to.
DEPTHCHANGE	Indicates an open window has been pushed or popped. Use `windepth` to determine the stacking order. The value returned is the window identification number of the window that the event pertains to.
WINSHUT	When queued, the window manager sends this token when the ***Close*** item is selected from a program's Window (frame) menu, or when the close fixture is selected from the title bar of a program's window. If WINSHUT is not queued, the Close item on the program's Window menu appears greyed-out, and has no effect if selected. The value returned is the window identification number of the window that the event pertains to.
WINQUIT	When queued, the window manager sends this token rather than killing a process when ***Quit*** is selected from a program's Window (frame) menu.

WINFREEZE/WINTHAW	If queued, the window manager sends these tokens when windows are stowed to icons and later unstowed, rather than blocking the processes of the stowed windows. These devices should be queued if the program plans to draw its own icon (see `iconsize`) or is a multi-window application.
INPUTCHANGE	Indicates a change in the input focus. If the value is 0, input focus has been removed from the process. If the value is non-0, it indicates the window number of the window that has just gained input focus.

5.5 Valuators

The following devices are valuators that return specific information about the system.

5.5.1 Timer Devices

The timer devices record an event every retrace interval. You can use a timer device to synchronize a graphics program with a real clock. To record events less frequently, use `noise`. For example, if you call:

```
noise (TIMER0, 30)
```

only every 30th event is recorded, so an event queue entry is made each half second.

5.5.2 Cursor Devices

The cursor devices are pseudo-devices that are equivalent to the valuators currently attached to the cursor. (See `attachcursor` for more information.)

5.5.3 Ghost Devices

Ghost devices, GHOSTX and GHOSTY, do not correspond to any physical devices, although they can be used to change a device under program control. For example, to drive the cursor from software, use `attachcursor(GHOSTX,GHOSTY)` to make the cursor position depend on the ghost devices. Then use `setvaluator` on GHOSTX and GHOSTY to move the cursor.

5.6 Controlling Peripheral Input/Output Devices

setvaluator

Valuators are single-value input devices. The value is a 16-bit integer. The horizontal and vertical motion of the mouse, or the turning of a dial, are valuators. `setvaluator` assigns an initial value *init* to a valuator. *min* and *max* are the minimum and maximum values the device can assume.

```
void setvaluator(val, init, min, max)
Device val;
short init, min, max;
```

In addition to routines that poll and queue input devices, there are routines that control the characteristics and behavior of the system's peripheral input/output devices. For example, some of these routines turn the keyboard click on and off (`clkon` and `clkoff`), or set the keyboard bell. You set these controls to your preference or needs.

clkon

`clkon` turns on the keyboard click.

```
void clkon()
```

clkoff

`clkoff` turns off the keyboard click.

```
void clkoff()
```

lampon

`lampon` and `lampoff` control the four lamps on the keyboard. Each 1 in the four lower-order bits of the *lamps* argument to `lampon` turns on the corresponding keyboard lamp.

```
void lampon(lamps)
char lamps;
```

lampoff

Each 1 in the four lower-order bits of the *lamps* argument to `lampoff` turns off the corresponding keyboard lamp.

```
void lampoff(lamps)
char lamps;
```

ringbell

`ringbell` rings the keyboard bell.

```
void ringbell()
```

setbell

`setbell` sets the duration of the keyboard bell: 0 is off, 1 is a short beep, and 2 is a long beep.

```
void setbell(mode)
char mode;
```

dbtext

`dbtext` writes text to the LED display in a dial and button box. The string *str* must be eight or fewer uppercase characters.

```
void dbtext(str)
string str;
```

setdblights

`setdblights` controls the 32 lighted switches on a dial and switch box. For example, to turn on switches 3 and 7, the third and seventh bits to the right of *mask* must be set to 1; that is, (1<<3)|(1<<7) .

```
void setdblights(mask)
unsigned_long mask;
```

5.7 Determining the Status of Video Options

You can determine and control the status of video options on your IRIS-4D Series system with the Graphics Library commands `getvideo`, `setvideo`, and `videocmd`. You can also determine information about the monitor type used on your system with the commands `getmonitor` and `getothermonitor`. The commands `setmonitor`, `blankscreen`, and `blanktime` allow you to specify operating parameters for the monitor on your system.

setvideo

`setvideo` sets the specified video hardware register, *reg*, to the specified *value*. *reg* expects the name of the register to access. *value* expects the value to be placed into *reg*. `setvideo` (and its counterpart, `getvideo`, described below) supports several different video boards (see Table 5-4).

Video Option Board	Register
Display Engine Board	DE_R1
CG2 Composite Video and Genlock Board	CG_CONTROL
	CG_CPHASE
	CG_HPHASE
	CG_MODE
VP1 Live Video Digitizer Board	VP_ALPHA
	VP_BRITE
	VP_CMD
	VP_CONT
	VP_DIGVAL
	VP_FBXORG
	VP_FBYORG
	VP_FGMODE
	VP_GBXORG
	VP_GBYORG
	VP_HBLANK
	VP_HEIGHT
	VP_HUE
	VP_MAPADD
	VP_MAPBLUE
	VP_MAPGREEN
	VP_MAPRED
	VP_MAPSRC
	VP_MAPSTROBE
	VP_PIXCNT
	VP_SAT
	VP_STATUS0
	VP_STATUS1
	VP_VBLANK
	VP_WIDTH

Table 5-4. setvideo and getvideo Register Values

The DE_R1 register is actually present only on the video board used in IRIS-4D/B/G/GT/GTX models. It is emulated on all other models.

The Live Video Digitizer is available as an option for IRIS-4D/GTX models only.

```
void setvideo(reg, value)
long reg, value;
```

getvideo

`getvideo` returns the value of the specified video hardware register. The returned value of `getvideo` is the one read from register *reg*, or -1. -1 indicates that *reg* is not a valid register or that you queried a video register on a system without that particular board installed. The legal values for *reg* are the same as those described in `setvideo`.

```
long getvideo(reg)
long reg;
```

videocmd

`videocmd` initializes the Live Video Digitizer peripheral board (optionally available for IRIS-4D/GTX systems only). You can initialize the Live Video Digitizer in either RGB or composite video mode, for both NTSC and PAL format video sources. `videocmd` takes one parameter, *cmd*, a command value that initiates the specified command sequence.

The accepted values for *cmd* are:

VP_INITNTSC_COMP	Initialize the optional Live Video Digitizer for a composite NTSC video source
VP_INITNTSC_RGB	Initialize the Live Video Digitizer for an RGB NTSC video source
VP_INITPAL_COMP	Initialize the Live Video Digitizer for a composite PAL video source
VP_INITPAL_RGB	Initialize the Live Video Digitizer for an RGB PAL video source

These symbolic constants are defined in the file *gl/vp1.h*.

```
void videocmd(cmd)
long cmd;
```

setmonitor

`setmonitor` sets the monitor to one of the following *types* (Table 5-5). Those constants are defined in the file *gl/get.h*.

Value of *type*	Monitor Type
HZ30	30 Hz interlaced
HZ30_SG	30 Hz noninterlaced with sync on green
HZ60	60 Hz noninterlaced
NTSC	NTSC
STR_RECT	Optional Stereo mode (1280 x 492 x 2)
PAL	PAL or SECAM

Table 5-5. Monitor Types

```
void setmonitor(type)
short type;
```

The DE_R1 register is actually present only on the video board used in IRIS-4D/B/G/GT/GTX models. It is emulated on all other models.

The Live Video Digitizer is available as an option for IRIS-4D/GTX models only.

```
void setvideo(reg, value)
long reg, value;
```

getvideo

`getvideo` returns the value of the specified video hardware register. The returned value of `getvideo` is the one read from register *reg*, or -1. -1 indicates that *reg* is not a valid register or that you queried a video register on a system without that particular board installed. The legal values for *reg* are the same as those described in `setvideo`.

```
long getvideo(reg)
long reg;
```

videocmd

`videocmd` initializes the Live Video Digitizer peripheral board (optionally available for IRIS-4D/GTX systems only). You can initialize the Live Video Digitizer in either RGB or composite video mode, for both NTSC and PAL format video sources. `videocmd` takes one parameter, *cmd*, a command value that initiates the specified command sequence.

blanktime

`blanktime` sets the screen blanking timeout. By default, the screen blanks (turns black) after the system receives no input for about 15 minutes. This protects the color display. `blanktime` changes the amount of time the system waits before blanking the screen. It can also disable the screen blanking feature.

nframes specifies the screen blanking timeout in frame times based on the standard 60 Hz monitor. For software compatibility, the factor of 60 is used, regardless of the monitor type. To calculate the value of *nframes*, multiply the desired blanking latency period (in seconds) by 60. For example, when *nframes* is 1800, the blanking latency period is 5 minutes. There are 60 frames per second; *nframes* is 60 times the number of seconds that the system waits before blanking the screen. When *nframes* is 0, screen blanking is disabled.

```
void blanktime(nframes)
long nframes;
```

5.8 Spaceball™ Devices

Table 5-6 lists the devices returned by `qread()` when the optional Spaceball input device sends an event onto the queue.

Devices	Description
SBPERIOD	Number of periods of 0.25 ms since sending the last non-0 set of Spaceball data
SBTX	Right/left push
SBTY	Up/down push
SBTZ	Away/towards push
SBRX	Twist about right/left axis
SBRY	Twist about up/down axis
SBRZ	Twist about away/towards axis
SBBUT1	Button 1
SBBUT2	Button 2
SBBUT3	Button 3
SBBUT4	Button 4
SBBUT5	Button 5
SBBUT6	Button 6
SBBUT7	Button 7
SBBUT8	Button 8
SBPICK	Pick button

Table 5-6. Spaceball Input Buttons

For more information about the optional Spaceball input device, see the manual *Using and Programming the Spaceball*, document number 007-6209-010.

6. Animation

Up to now, all the programs draw single images on the screen. Using the techniques presented so far, it is possible to make only the simplest images appear to move smoothly. This chapter introduces a technique called *double buffering*, which allows you to create graphics that appear to change smoothly.

6.1 Double Buffering

The IRIS-4D Series workstation lets you address frame buffer memory in either of two modes:

- *single buffer* mode, in which a program addresses frame buffer memory as a single buffer whose pixels are always visible
- *double buffer* mode, in which a program addresses frame buffer memory as if it were two buffers, only one of which is displayed at a time

By default, the system is in single buffer mode. Whatever you draw into the bitplanes is immediately visible on the screen. For static drawings, this is acceptable, but it does not provide smooth animated motion. If you try to animate a drawing in single buffer mode, you can see a visible flicker in all but the simplest drawing operations.

For smooth motion, it is preferable to display a completely drawn image for a certain time (for instance, a few 60ths of a second), then present the next frame, also completely drawn, during the next time period, and so on.

Double buffering provides this capability. The system's standard bitplanes are divided into two halves; one half is displayed while the other half is rendering. When the drawing is complete, the system swaps buffers, the previously invisible buffer (now containing the next frame) becomes visible, and the previously visible buffer becomes invisible and becomes available for drawing the next frame.

The currently visible buffer is called the front buffer and the invisible, drawing buffer is the back buffer. Double buffering works in either RGB mode or color map mode.

6.2 Double Buffer Mode

doublebuffer

`doublebuffer` sets the display mode to double buffer mode. It does not take effect until you call `gconfig`. In double buffer mode, the bitplanes are partitioned into two groups, the front bitplanes and the back bitplanes. In double buffer mode, only the front bitplanes are displayed, and drawing routines normally update only the back bitplanes; `frontbuffer` and `backbuffer` can override the default. `gconfig` sets `frontbuffer` to FALSE and `backbuffer` to TRUE in double buffer mode.

```
void doublebuffer()
```

swapbuffers

The display hardware in the system constantly reads the contents of the visible buffer (the front buffer in double buffer mode), and displays those results on the screen. On a standard monitor, the electron guns sweep from the top of the screen to the bottom, refreshing all pixels, 60 times each second. If the graphics hardware changes the contents of the visible frame buffer, the next time the refresh hardware reads a changed pixel, the system draws the new value instead of the old one.

After sweeping out the entire frame, the guns are reset to the top of the screen. This takes a short time, called the vertical retrace, or screen refresh, and during this period (much shorter than 60th of a second), nothing can change on the screen. `swapbuffers` exchanges the front and back buffers in double buffer mode during the next vertical retrace.

After the entire frame is rendered into the back buffer, `swapbuffer` is called to make it the visible buffer. The `swapbuffer` subroutinewaits for the next screen refresh before actually swapping the front and back buffer.s If it did not happen, a frame would be drawn partly in one buffer, and partly in the other, causing a serious visual disturbance. Because screen refresh occur every 60th of a second on the standard monitor, `swapbuffers` can block the running process for up to that long. (The default monitor is refreshed 60 times per second. Other options can have other retrace periods. See `setmonitor` and `getmonitor`.)

```
void swapbuffers()
```

Because `swapbuffer` blocks the user program until the next screen refresh (1/60 sec), every frame takes *n* screen refreshes to render and display, where *n* is the ceiling function of the actual rendering time. For example, if a scene takes 1.9 refreshes to render, then every frame takes 2 refreshes to render and display. Therefore, the application performs at 60/2 or 30 frames per second. If you add another polygon to the scene, and it now takes 2.1 refreshes to render, or 3 refreshes to render and display, the frame rate drops from 30 to 60/3 or 20 frames per second. There is no smooth degradation. Properly tuning such a program can be tricky if you require smooth motion. Keep in mind that while the geometry is moving about, the time it takes to render each frame varies.

The `swapbuffer` call is ignored in single buffer mode.

mswapbuffers

On IRIS-4D/VGX Series systems, both overlay and underlay planes can also be double buffered. The `mswapbuffers` subroutine lets you swap any combination of the available frame buffers at the same time.

```
void mswapbuffers(fbuf)
long fbuf;
```

The constants you can use as the value of *fbuf* are:

NORMALDRAW	Swap the front and back buffers of the normal color bitplanes
OVERDRAW	Swap the front and back buffers of the overlay bitplanes
UNDERDRAW	Swap the front and back buffers of the underlay bitplanes

These constants can be bit-ORed together to swap multiple buffers simultaneously. For example, to swap front and back for the normal frame buffer and for the underlay planes, include the following line in your program:

```
mswapbuffers(NORMALDRAW | UNDERDRAW);
```

Just like `swapbuffers`, `mswapbuffers` blocks until the next vertical retrace period. It is ignored by frame buffers that are not in double buffer mode.

gconfig

Before drawing anything, you must set the frame buffers into the correct configuration, such as colorindex or RGB mode. `The gconfig` subroutine performs this function. Configuring the frame buffer is a two-step process:

1. You must indicate how to configure the frame buffer.

2. You must call `gconfig` to set the system into that particular configuration.

The following configurations are available: `acsize`, `cmode`, `doublebuffer`, `multimap`, `onemap`, `overlay`, `RGBmode`, `singlebuffer`, `stensize`, and `underlay`. These subroutines must be followed by `gconfig` to take effect.

After a `gconfig` call, writemask and color are no longer defined. The contents of the color map do not change.

```
void gconfig()
```

The following program illustrates the use of double buffer mode. It draws a set of balls bouncing around a square, starting at random positions and going at random speeds:

```
#include <gl/gl.h>
#define MAXBALLS 1000

float v0[] = {-1.05, -1.05};
float v1[] = {-1.05, 1.05};
float v2[] = {1.05, 1.05};
float v3[] = {1.05, -1.05};

long balls;
float xpos[MAXBALLS], ypos[MAXBALLS], dx[MAXBALLS];
float dy[MAXBALLS];
long col[MAXBALLS];

float randfloat()
{
    float numerator, denominator;
    numerator = rand()&0x7fff;
    denominator = 0x7fff;
    return numerator/denominator;
}

main(argc, argv)
int argc;
char **argv;
{
    long i;

    if (argc != 2) {
        printf("usage: %s <ball count>\\n", argv[0]);
        exit(1);
    }
    balls = atoi(argv[1]);

    for (i = 0; i < balls; i++) {
        xpos[i] = randfloat();
        ypos[i] = randfloat();
        dx[i] = randfloat()/50.0;
        dy[i] = randfloat()/50.0;
        col[i] = rand()&0xffffff;
    }
```

```
    keepaspect(1, 1);
    winopen("bounce");
    doublebuffer();
    RGBmode();
    gconfig();
    ortho2(-1.1, 1.1, -1.1, 1.1);
    while (1) {
        for (i = 0; i < balls; i++) {
            xpos[i] = xpos[i] + dx[i];
            ypos[i] = ypos[i] + dy[i];
            if ((xpos[i] >= 1.0) || (xpos[i] <= -1.0))
                  dx[i] = -dx[i];
            if ((ypos[i] >= 1.0) || (ypos[i] <= -1.0))
            dy[i] = -dy[i];
        }
        cpack(0);
        clear();
        cpack(0xffffff);
        bgnclosedline();
        v2f(v0); v2f(v1); v2f(v2); v2f(v3);
        endclosedline();
        for (i = 0; i < balls; i++) {
            cpack(col[i]);
            circf(xpos[i], ypos[i], .05);
        }
        swapbuffers();
    }
}
```

singlebuffer

In single buffer mode, the system simultaneously updates and displays the image data in the active bitplanes; consequently, incomplete or changing pictures can appear on the screen. `singlebuffer` does not take effect until `gconfig` is called. Single buffer mode is the default.

```
void singlebuffer()
```

frontbuffer

Note: The subroutine calls contained in the remainder of this chapter might be considered advanced topics. You do not need to use these calls unless you are tuning your program for maximum performance.

Sometimes in double buffer mode, it is useful to be able to write the same thing into both buffers at once. For example, suppose an animated image has both a fixed part and a changing part. The fixed part needs to be drawn only once, but into both buffers. It is most easily done by enabling the front buffer (as well as the back buffer) for writing, drawing the image, and then disabling the front buffer. The animation then proceeds by drawing the changing part of the image using the usual double buffering techniques. `frontbuffer(TRUE)` in double buffer mode enables simultaneous updating of (or writing into) both the front and the rear buffers. Its argument is a Boolean value.

When the value of *b* is FALSE (the default value), the front buffer is not enabled for writing. When the value of *b* is TRUE, the front buffer is enabled for writing. This routine is useful only in double buffer mode.

`gconfig` sets `frontbuffer` to FALSE.

```
void frontbuffer(b)
Boolean b;
```

backbuffer

It is sometimes convenient to update both the front and the back buffers, or to update the front buffer instead of the back. `backbuffer` enables updating in the back buffer. Its argument is a Boolean value. When the value of *b* is TRUE, the default, the back buffer is enabled for writing. When the value of *b* is FALSE, the back buffer is not enabled for writing.

`gconfig` sets `backbuffer` to TRUE.

```
void backbuffer(b)
Boolean b;
```

getbuffer

`getbuffer` indicates which buffer(s) are enabled for writing in double buffer mode. The returned values operate as a bitmask (that is, the number returned represents the numeric value of all the bits currently set). The default, 1, means the back buffer is enabled (as does any odd value); 2 means that the front buffer is enabled (or any value in which the 2 bit is set); and 3 (or value in which the 3 bit is set) means that both are enabled. `getbuffer` returns zero if neither buffer is enabled or if the system is not in double buffer mode. If the *z*-buffer (see Chapter 8) is enabled for drawing, `getbuffer` can return 4, 5, 6, or 7.

```
long getbuffer()
```

Each of the possible values also has an associated symbolic value that you can use in the call to `getbuffer` (Table 6-1):

Value	Buffer Enabled	Symbolic Name
0	none	NOBUFFER
1	back buffer	BCKBUFFER
2	front buffer	FRNTBUFFER
3	both buffers	BOTHBUFFERS
4	z buffer drawing	DRAWZBUFFER
5	z buffer plus back buffer	
6	zbuffer plus front buffer	
7	z buffer plus both buffers	

Table 6-1. Symbolic Values for getbuffer Statement

swapinterval

`swapinterval` defines a minimum time between buffer swaps. For example, a swap interval of 5 refreshes the screen at least five times between execution of successive calls to `swapbuffer`. `swapinterval` is typically used when you want to show frames at a constant rate, but the images vary in complexity. To achieve a constant rate, set the swap interval long enough that even the most complex frame can be drawn in that time. For drawing a simple frame, the user's process simply blocks and waits until the swap interval is used up. The default interval is 1.

`swapinterval` is valid only in double buffer mode.

```
void swapinterval(i)
short i;
```

getdisplaymode

`getdisplaymode` returns the current display mode. 0 indicates RGB single buffer mode; 1 indicates single buffer mode; 2 indicates double buffer mode; 5 indicates RGB double buffer mode. Modes 3 and 4 are unused.

```
long getdisplaymode()
```

gsync

`gsync` pauses execution until a vertical retrace occurs. It was intended as a method to synchronize drawing with the vertical retrace to achieve animation in single buffer mode. It does not work well. Some users also called `gsync` a number of times to get short time delays (since each call waited until the next 60th of a second). Use the system call `sginap` instead.

`gsync` is also used for rubberbanding in single buffer mode.

`gsync` is included primarily for compatibility with systems that might not have enough bitplanes to use double buffer mode. The IRIS-4D/GT always has two complete sets of color bitplanes. The Personal IRIS with 24 bits per pixel standard, permits double buffering by using two buffers with 12-bit pixels. `gsync` waits for the next vertical retrace period. Due to pipeline and operating system delays, smooth motion in single buffer mode is often impossible. `gsync` should be used only as a last resort, and it might not work.

```
void gsync()
```

7. Coordinate Transformations

When displaying 3-D shapes, it is useful to be able to move the shapes around relative to one another and to the viewer, to rotate and scale them, and to be able to change the viewer's point of view, field of view, and orientation. The subroutines in this chapter allow you to manipulate geometric figures and viewpoints in 3-D space in general ways.

Three-dimensional transformations tend to be difficult to describe, and there are many ways to visualize and think about them. Do not be discouraged if the subject is not perfectly clear after one reading. Reread the chapter, and try experiments with the sample code segments included here. Try writing your own programs. This chapter provides only one viewpoint; other books on computer graphics present the subject in slightly different ways that may be clearer to you.

To understand how IRIS-4D Series systems convert 3-D coordinates of geometric figures into pixels on the screen, consider these two sets of operations:

- A set of 3-D operations, such as rotation, translation, and scaling, that move the objects and viewpoint to the desired position for a given scene
- An operation that maps 3-D points to 2-D screen coordinates, taking into consideration the portion of 3-D space (as well as its orientation with respect to the screen) that is visible during a given scene

The 3-D operations can be further divided into projection, viewing, and modeling transformations. Conversion from the original 3-D figures to the 2-D pixels on the screen is handled by another set of subroutines, including `viewport` and `lsetdepth`.

There are also subroutines to save and restore transformations that can be useful in a hierarchical display structure, which is extremely convenient for 3-D graphics.

7.1 Coordinate Systems

Figure 7-1 illustrates the coordinate systems used at different stages of the drawing process. The coordinate systems begin with a 3-D system defined in right-handed cartesian floating point coordinates—in other words, vertices are specified in *(x, y, z)* triplets. Call this the *object coordinate system.* There are no limits to the size of sensible coordinates (other than the largest legal floating point value).

The *eye coordinate system* is the result of object coordinates transformed by the modeling and viewing (or ModelView) matrix. The eye coordinate system is the system in which lighting calculations are performed internally. When the IRIS transforms objects expressed in eye coordinates by the Projection matrix, the output is expressed in *clip coordinates.* It is in this coordinate system that values returned by a call to `getgpos` are expressed.

The next system is called the *normalized coordinate system.* It is discussed in much more detail later. Clip coordinates are converted to normalized coordinates by first limiting x, y, and z to the range $-w \leq x,y,z \leq w$ (clipping), then dividing x, y, and z by w. The result is normalized coordinates in the range $-1 \leq x,y,z \leq 1$. The space of normalized coordinates is called the *3-D unit cube.*

The x and y coordinates of this 3-D unit cube are scaled directly into the next coordinate system, usually called the *window coordinate system.* Depth-cueing occurs in the window coordinate system. If you draw into an arbitrarily placed window on the screen, the pixel at the lower-left corner of the window has window coordinates (0,0). These window coordinates, once modified by a window offset that represents the window's location on the screen, represent the *screen coordinate system,* which corresponds to pixel values.

Screen coordinates are typically thought of as 2D, but in fact all three dimensions of the normalized coordinates are scaled, and there is a screen z coordinate that can be used for many things, such as hidden surface removal or depth cueing.

All coordinate systems, including window and screen, are continuous. All current IRIS-4D Series models represent coordinate values in eye, clip, and normalized coordinates with single-precision, floating point numbers. All accept object coordinate values as 16-bit integers, 32-bit integers (with signed 24-bit range), and single- and double-precision floating point numbers. The format and precision of window and screen coordinates vary from model to model, and as a function of the GL `subpixel` mode. These coordinates are *not* limited to integer values, however.

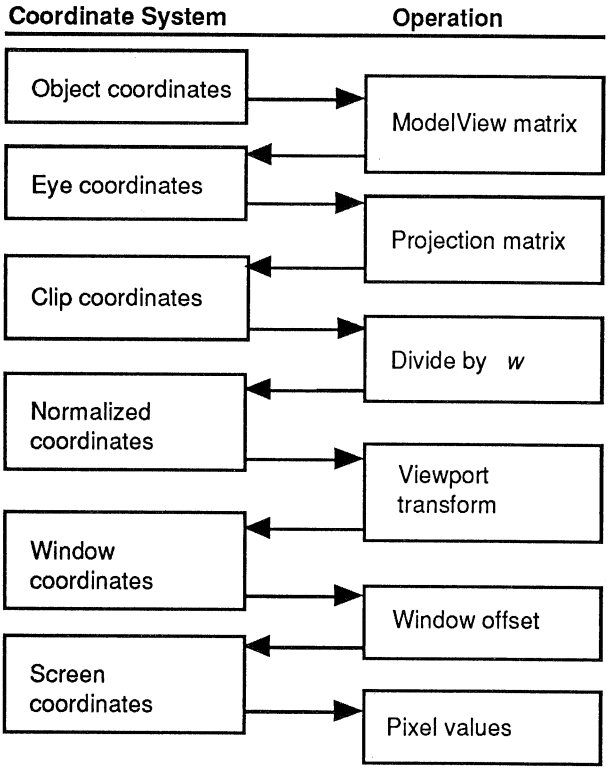


Figure 7-1. Coordinate Systems

A note on "world coordinates": If the ModelView matrix were separated into a Model matrix and a View matrix, the coordinate system between these matrices would be correctly referred to as the world coordinate system. Because the GL concatenates modeling and viewing transformations on a single matrix (ModelView), there are no world coordinates.

By default, the Graphics Library combines the ModelView and the Projection matrix into a single "transformation" matrix. Because this single-matrix operation mode does not support many of the advanced features of the GL, it is not described in this chapter. Rather, this chapter discusses only the multimatrix mode, in which separate ModelView and Projection matrices are maintained.

7.2 Projection Transformations

Viewing items in perspective on the computer screen is like looking through a rectangular piece of perfectly transparent glass at the items. If you draw a line from your eye through the glass until it hits the item, imagine coloring a dot on the glass where the line passes through the same color as the item that the line intersects. If this were done for all possible lines through the glass, the coloring were perfect, and the eye not allowed to move, then the picture painted on the glass would be indistinguishable from the true scene.

The collection of all the lines leaving your eye and passing through the glass would form an infinite four-sided pyramid with its apex at your eye. Anything outside the pyramid would not appear on the glass, so the four planes passing through your eye and the edges of the glass would clip out invisible items. These are called the left, right, bottom, and top *clipping planes*. The geometry hardware also provides two other clipping planes that eliminate anything too far from or too near the eye. They are called the *near* and *far* clipping planes. Near and far clipping is always turned on, but it is possible to set the near plane very close to the eye and/or the far plane very far from the eye so that all the geometric items of interest are visible.

Because floating point calculations are not exact, it is a good idea to move the near plane as far as possible from the eye, and to bring in the far plane as close as possible. This gives optimal resolution for distance-based operations such as hidden surface removal and depth-cueing, discussed in later chapters.

Thus, for a perspective view, the visible region of the world looks like an Egyptian pyramid with the top sliced off. The technical name for this is a *frustum*, or *rectangular viewing frustum.*

perspective

`perspective` defines a Projection matrix that maps a frustum of eye coordinates so that it exactly fills the unit cube (after x, y, and z are each divided by w). The particular frustum so mapped is part of a pyramid whose apex is at the origin (0.0, 0.0, 0.0). The base of the pyramid is parallel to the x-y plane, and it extends along the negative z axis. In other words, it is the view obtained with the eye at the origin looking down the negative z axis, and the plate of glass is perpendicular to the line of sight.

`perspective` has four arguments: the field of view in the y direction, the aspect ratio, and the distances to the near and far clipping planes. The field of view is an angle made by the top and bottom clipping planes that is measured in tenths of degrees (so 900 is a right angle). The aspect ratio is the ratio of the x dimension of the glass to its y dimension. It is a floating point number. For example, if it is 2.0, the glass is twice as wide as it is high. Typically, you choose the aspect ratio so that it is the same as the aspect ratio of the window on the screen, but it need not be. The distances to the near and far clipping planes are floating point values.

The following program illustrates a simple use of `perspective`:

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

float v0[3] = {-3.0, 3.0, 0.0};
float v1[3] = {-3.0, -3.0, 0.0};
float v2[3] = {2.0, -3.0, -4.0};
float v3[3] = {2.0, 3.0, -4.0};

main()
{
  keepaspect(1, 1);
  winopen("perspective");
  mmode(MVIEWING);
  perspective(900, 1.0, 2.0, 5.0);
  RGBmode();
  gconfig();
  cpack(0x0);
  clear();
  bgnpolygon();
  cpack(0xff); v3f(v0); v3f(v1);
  cpack(0xff00); v3f(v2); v3f(v3);
  endpolygon();
  sleep(20);
}
```

This program draws a single rectangle whose shade varies from bright red at z=0.0 to bright green at z=-4.0. Because the eye is at (0.0, 0.0, 0.0), the rectangle recedes into the distance, and because the field of view is 90 degrees and the near clipping plane is at 2.0, the rectangle is clipped by the top, bottom, and near clipping planes. Because the rectangle is clipped by the near clipping plane, the visible part of the polygon nearest the eye is not bright red, but already looks yellowish. If you bring in the near clipping plane, the near end of the polygon looks more and more red. Figure 7-2 shows a top view. The heavy line is the rectangle.

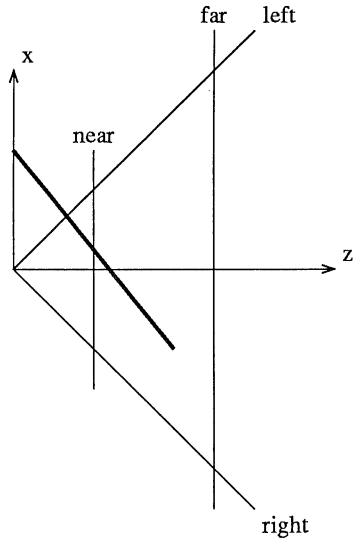


Figure 7-2. The perspective Subroutine

`keepaspect` tells the window manager to assure that the window has the *x* and *y* dimensions in a 1 to 1 ratio, i.e., a square. An equally accurate picture could be made by setting a 2 to 1 ratio, but then the aspect ratio in `perspective` would have to be changed to 2.0. You might try this. Also try varying other parameters of `perspective` —change the field of view and the near and far clipping planes to see the effects.

In a real application, you probably want to match the aspect ratio of `perspective` to the aspect ratio of the window when you sweep out a window of arbitrary shape and size. See the window manager documentation on `getsize` to learn how to find the shape of the current window.

All the projection transformations work basically the same way. A viewing volume is mapped into the unit cube, the geometry outside the cube is clipped out, and the remaining data is linearly scaled to fill the window (actually the *viewport*, which is discussed later). The only differences between the projection transformations are the definitions of the viewing volumes.

```
void perspective(fovy, aspect, near, far)
Angle fovy;
float aspect;
Coord near, far;
```

window

Another projection transformation that shows the world in perspective is `window`. It is similar to `perspective`, but its viewing frustum is defined in terms of distances to the left, right, bottom, top, and near and far clipping planes.

```
void window(left, right, bottom, top, near, far)
Coord left, right, bottom, top, near, far;
```

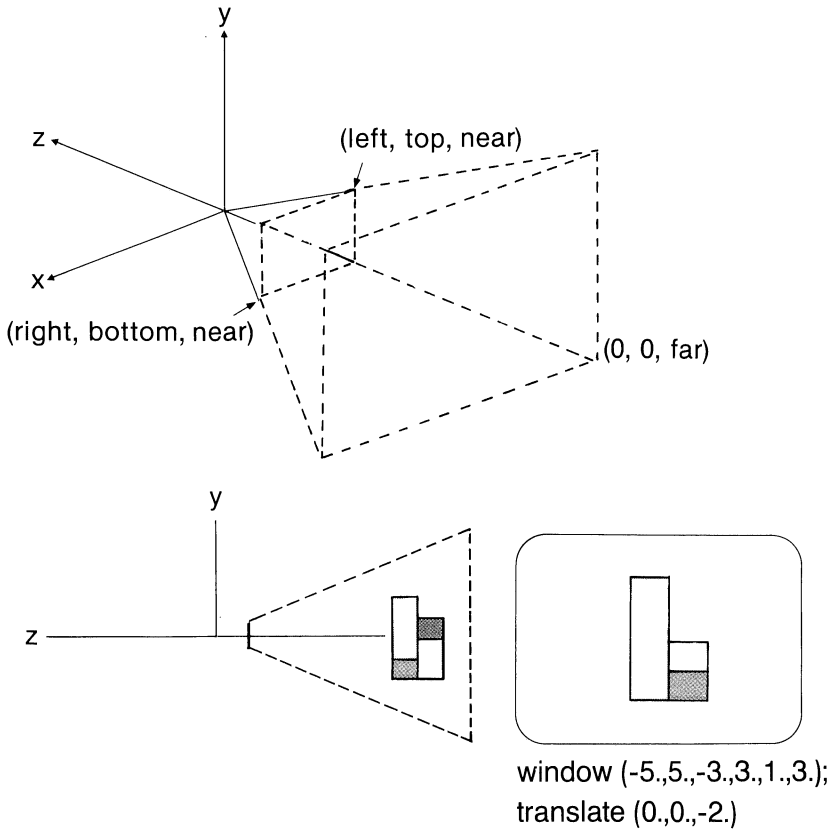
Because window allows separate specifications at all six surfaces of the viewing frustum, it can be used to specify asymmetric volumes. These are useful in special circumstances, such as multiple view simulations, and for special operations, such as antialiasing using the accumulation buffer (see Chapter 15).

`window` specifies the position and size of the rectangular viewing frustum closest to the eye (in the near clipping plane), and the location of the far clipping plane. `window` projects the image onto the screen with perspective (see Figure 7-3).

The other two projection subroutines that are part of the Graphics Library are the orthogonal transformations. Their viewing volumes are rectangular parallelepipeds (rectangular boxes). They correspond to the limiting case of a perspective frustum as the eye moves infinitely far away and the field of view decreases appropriately.

Another way to think of the `ortho` subroutines is that the geometry outside the box is clipped out, and then the geometry inside is projected parallel to the z axis onto a face parallel to the x-y plane.

`ortho` allows you to specify the entire box—the x, y, and z limits. `ortho2` requires a specification of only the x and y limits. The z limits are assumed to be -1 and 1. `ortho2` is usually used for 2-D drawing, where all the z coordinates are zero. It is really a 3-D transformation, and if you use `ortho2` and try to draw objects with z coordinates outside the range $1.0 \leq z \leq 1.0$, they will be clipped out.



window defines a viewing window in the x-y plane looking down the -z axis. A perspective view of the image is projected onto the window.

Figure 7-3. The window Subroutine

ortho

`ortho` defines a box-shaped enclosure in the eye coordinate system. *left*, *right*, *bottom*, and *top* define the *x* and *y* clipping planes. *near* and *far* are distances along the line of sight and can be negative. In other words, the *z* clipping planes are located at *z* = -near and *z* = -far. Figure 7-4 shows an example of a 3-D orthographic projection.

```
void ortho(left, right, bottom, top, near, far)
Coord left, right, bottom, top, near, far;
```

ortho2

`ortho2` defines a 2-D clipping rectangle. Because of the transformation of floating point values to integers as the coordinate systems are modified to use screen coordinates, and because of the need to align lines on pixels rather than on interpixel boundaries, the `ortho2` statement in the following fragment defines the correct clipping rectangle for the window created with the corresponding `prefposition` statement:

```
prefposition(100, 500, 100, 500);
ortho2(99.5, 500.5, 99.5, 500.5);
```

This causes the clipping rectangle to clip only items that are completely within the window defined by the `prefposition` statement. These two statements ensure that the `ortho2` statement clips on (and point-samples within) boundaries that are completely visible within the window defined by the `prefposition` statement.

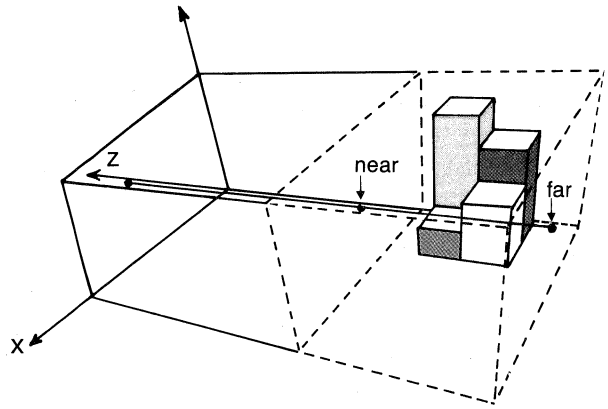
```
void ortho2(left, right, bottom, top)
Coord left, right, bottom, top;
```

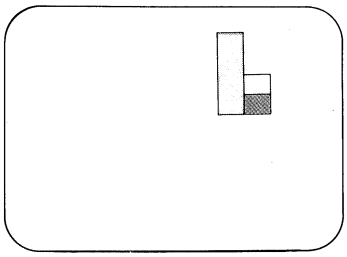
Caution: Because `ortho` allows separate specification of the left, right, bottom, and top clipping planes (rather than just width and height of the viewing volume), it can move the effective viewpoint off the positive *z* axis. (Recall that both `perspective` and `window` force the viewpoint to the origin.) Thus, `ortho` incorporates some aspects of a viewing transformation, as well as of a projection transformation. Be careful when using `ortho` with asymmetric volume boundaries that you do not duplicate this viewpoint offset in your viewing transformation (see Section 7.3).

7.3 Viewing Transformations

All the projection transformations discussed so far have assumed that the eye is at least looking toward the negative *z* axis, and the two perspective subroutines actually assume that the eye is at the origin. For the orthogonal transformations, it does not make sense to talk about the exact position of the eye, only about the direction it is looking.

The viewing transformations allow you to specify the position of the eye in the world coordinate system, and to specify the direction toward which it is looking. `polarview` and `lookat` provide convenient ways to do this. `polarview` assumes that the object you are viewing is near the origin, and the eye's position is specified by a radius (distance from the origin) and by angles measuring the azimuth and elevation. The specification is similar to polar coordinates; hence, the name. There is still one degree of freedom, because these values tell only where the eye is relative to the object. A twist argument tells what direction is up.





ortho (-5.,5.,-3.,3.,1.,3.);
translate (0.,0.,-2.);

ortho defines a viewing window in the x-y plane, looking down the -z axis. An orthographic view of the object between the near and far planes is projected onto the window.

Figure 7-4. The ortho Subroutine

`lookat` allows you to specify the eye's position in space and a point at which you are looking. Both points are given with cartesian *x*, *y*, and *z* coordinates. Again, a twist argument is required to eliminate the last degree of freedom. Note that once you specify the eye position, the point you are looking at could be any point along a line, and the identical transformation is specified.

Both viewing subroutines work with a projection subroutine. If you want to view the point (1, 2, 3) from the point (4, 5, 6) in perspective, use `perspective` with `lookat` as described below. When the orthogonal projections are used, the exact position of the eye used in the viewing subroutines does not make a difference. All that matters is the viewing line of sight.

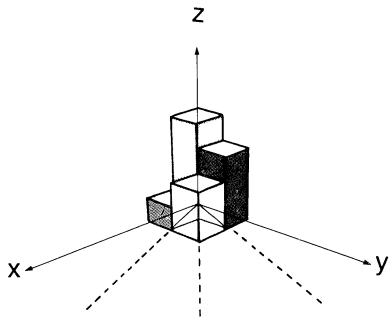
The viewing transformations work mathematically by transforming, via rotations and translations, the position of the eye to the origin and the viewing direction so that it lies along the negative z axis.

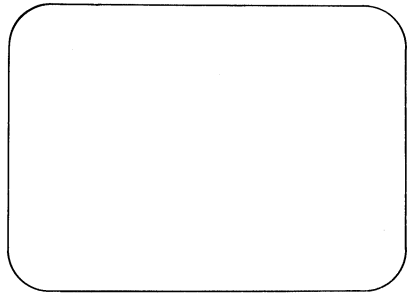
polarview

`polarview` defines the viewer's position in polar coordinates. The first three arguments, *dist*, *azim*, and *inc*, define a viewpoint. *dist* is the distance from the viewpoint to the world space origin. *azim* is the azimuthal angle in the *x*-*y* plane, measured from the *y* axis. *inc* is the incidence angle in the *y*-*z* plane, measured from the *z* axis. The line of sight is the line between the viewpoint and the world space origin.

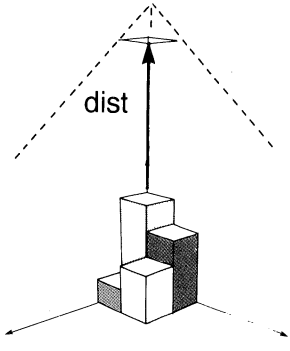
twist rotates the viewpoint around the line of sight using the right-hand rule. All angles are specified in tenths of degrees and are integers. Figure 7-5 shows examples of `polarview` .

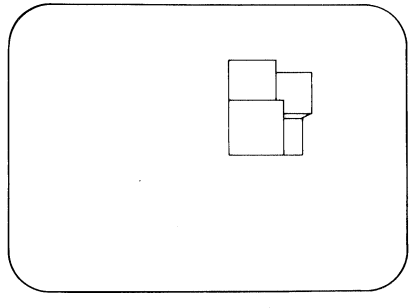
```
void polarview(dist, azim, inc, twist)
Coord dist;
Angle azim, inc, twist;
```





polarview(0.,0,0,0);





polarview(10.,0,0,0);

polarview has four arguments: the viewing distance from the origin, an incidence angle measured from the z-axis in the y-z plane, an azimuthal angle measured from the y-axis in the x-y plane, and a twist around the line of sight. Each frame shows the viewpoint and viewed image as additional arguments to *polarview* are supplied.

Figure 7-5. The polarview Subroutine

lookat

`lookat` defines a viewpoint and a reference point on the line of sight in world coordinates. The viewpoint is at (*vx*, *vy*, *vz*)and the reference point is at (*px*, *py*, *pz*). These two points define the line of sight. *twist* measures right-hand rotation about the *z* axis in the eye coordinate system. Figure 7-6 illustrates `lookat` .

```
void lookat(vx, vy, vz, px, py, pz, twist)
Coord vx, vy, vz, px, py, pz;
Angle twist;
```

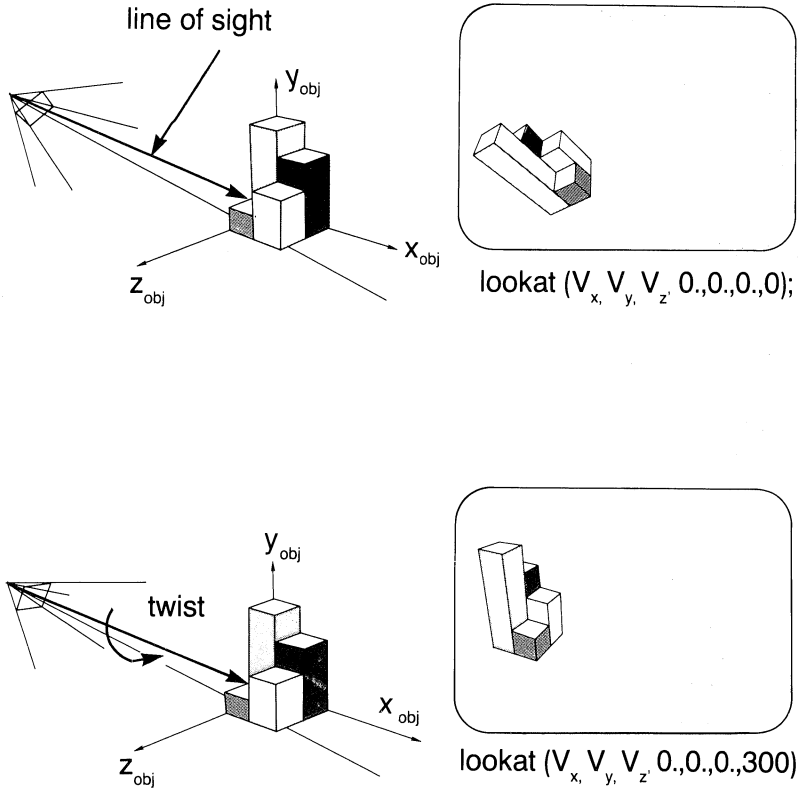
The following program illustrates how to use a viewing transformation with a projection transformation.

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

long v0[3] = {-1, -1, -1};
long v1[3] = {-1, -1, 1};
long v2[3] = {-1, 1, 1};
long v3[3] = {-1, 1, -1};
long v4[3] = {1, -1, -1};
long v5[3] = {1, -1, 1};
long v6[3] = {1, 1, 1};
long v7[3] = {1, 1, -1};
```

```
main()
{
  keepaspect(3, 2);
  winopen("lookat");
  mmode(MVIEWING);
  perspective(900, 1.5, .1, 10.0);
  lookat(5.0, 4.0, 6.0, 1.0, 1.0, 1.0, 0);
  color(BLACK);
  clear();
  color(WHITE);
  bgnline();
  v3i(v0); v3i(v1); v3i(v2); v3i(v3);
  v3i(v0); v3i(v4); v3i(v5); v3i(v6);
  v3i(v7); v3i(v4); v3i(v5); v3i(v1);
  v3i(v2); v3i(v6); v3i(v7); v3i(v3);
  endline();
  sleep(3)
  gexit();
  exit(0);
}
```

This program draws a wire-frame cube centered at the origin. Notice that the window has an aspect ratio of 3:2, or 1.5:1, so the aspect ratio of `perspective` is 1.5 to match. `lookat` looks from the point (5.0, 4.0, 6.0) at the corner (1.0, 1.0, 1.0) of the cube, so that corner appears centered in the window.

The near and far clipping planes are at .1 and 10.0. Nothing is clipped out on the near end but the far corner of the cube is about 10.49 away from the eye, so the far clipping plane clips a bit of the corner. Try modifying various aspects of the above program and notice their effects.



lookat defines a viewpoint, a reference point along the line of sight, and a twist angle. The top illustrations show the viewer and viewed image with no twist; twist is added to the lower illustrations.

Figure 7-6. The lookat Subroutine

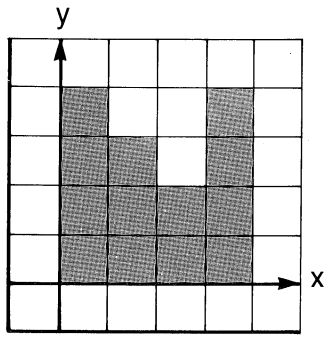
7.4 Modeling Transformations

When you create a graphical object, or geometric model, the system creates it with respect to its own coordinate system. You can manipulate the entire object using the modeling transformation subroutines: `rotate`, `rot`, `translate`, and `scale`. By combining or linking drawing subroutines, you can create more complex modeling transformations that express relationships among different parts of a complex object.

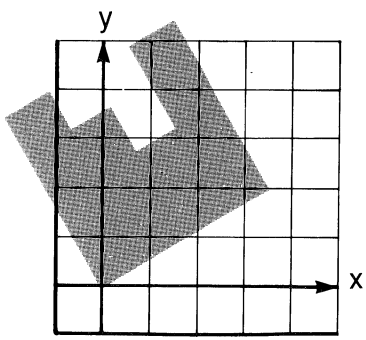
rotate

`rotate` rotates graphical objects; it specifies an angle and an axis of rotation. The angle is given in tenths of degrees according to the right-hand rule, which is as follows: as you look down the positive rotation axis to the origin, positive rotation is counterclockwise. A character, either *x*, *y*, or *z*, defines the axis of rotation. (The character can be upper- or lowercase.) For example, the object shown in Figure 7-7(a) is rotated 30 degrees with respect to the *y* axis in Figure 7-7(b). All objects drawn after `rotate` executes are rotated. This means that you must pay close attention to the order in which you specify transformation operations, or your program might provide you with some surprising results. The ordering of transformations is discussed later in this chapter.

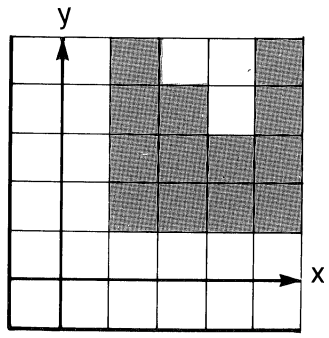
```
void rotate(a, axis)
Angle a;
char axis;
```



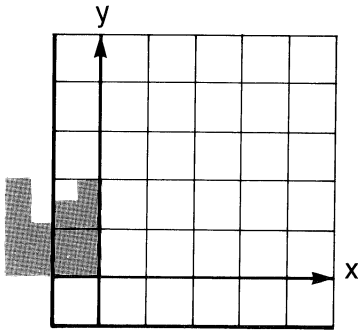
(a) original object at (0,0,0)



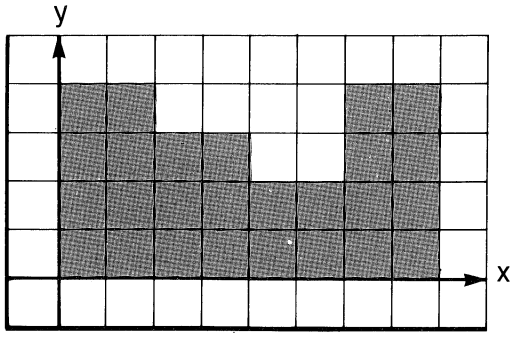
(b) rotate (300 ,'Z');



(c) translate (1.,1.,0.);



(d) scale (–.5,.5,1.);



(e) scale (2.,1.,1.);

The modeling routines are *rotate, translate,* and *scale.* The object shown in (a) is rotated in (b), translated in (c), and scaled in (d) and (e).

Figure 7-7. Modeling Commands

rot

`rot` is the same as `rotate` ; it specifies an angle and an axis of rotation in floating point values. The angle is measured in degrees.

```
void rot(a, axis)
float a;
char axis;
```

translate

`translate` moves the object origin to the point specified in the current object coordinate system. The object in Figure 7-7(a) is translated in Figure 7-7(c). All objects drawn after `translate` executes are translated.

```
void translate(x, y, z)
Coord x, y, z;
```

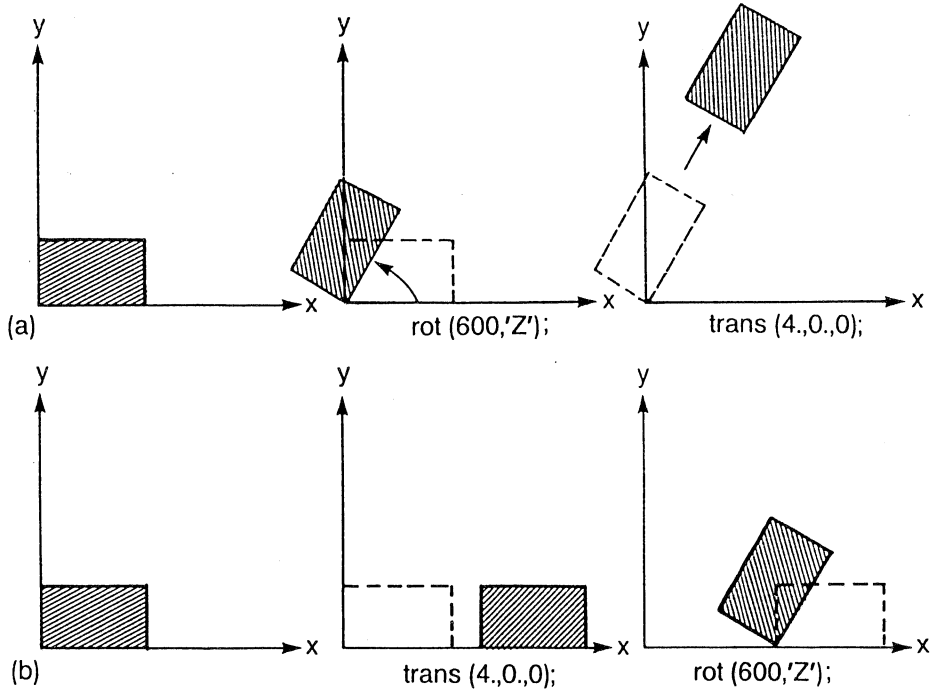
scale

`scale` shrinks, expands, and mirrors objects. Its three arguments (x, y, z) specify scaling in each of the three coordinate directions. Values with magnitude of more than 1 expand the object; values with magnitudes of less than 1 shrink it. Negative values cause mirroring.

All objects that are drawn after `scale` executes are scaled. The object shown in Figure 7-7(a) is shrunk to one-quarter of its original size and is mirrored about the y axis in Figure 7-7(d). It is scaled only in the x direction in Figure 7-7(e).

You can combine `rotate`, `rot`, `translate`, and `scale` to produce more complicated transformations. The order in which you apply these transformations is important. Figure 7-8 shows two sequences of `translate` and `rotate`; each sequence has different results.

```
void scale(x, y, z)
float x, y, z;
```



The modeling routines are not commutative: if you reverse the order of operations, you can get different results. (a) shows a rotation of 60 degrees about the origin followed by a translation of 4 degrees in the x-direction. (b) shows the same operations performed in the reverse order. Note that rotations are about the origin of the coordinate system.

Figure 7-8. The translate and rotate Subroutines

7.5 Controlling the Order of Transformations

Each time you specify a transformation such as rotate or translate, the software automatically generates a *transformation matrix* that specifies the amount by which the object's coordinate system is to be rotated or translated. The current transformation matrix is then premultiplied by the generated matrix, effecting the desired transformation. The actual transformations are done in an order opposite to that specified. In other words, you specify the viewing matrix first, followed by the modeling transformations, so that vertices are first positioned correctly in world coordinates, then the eye point moves to the origin looking down the negative *z* axis.

The reason for the reverse order is that the transformations are accomplished in the hardware by matrix multiplication, and historically, the matrix multiplication hardware allows only left multiplications. Thus, a vector *v*, transformed by a modeling transformation *M* and a viewing transformation *V* (in that order), undergoes the following multiplications:

$$v \rightarrow vM \rightarrow (vM)V = vMV$$

The hardware concatenates modeling and viewing transformations onto one matrix to save time, but because it performs multiplication only on the left, it must start with *V*, then generate *MV*. Because the Projection matrix is stored separately from the ModelView matrix, it does not matter whether projection is specified before or after the modeling and viewing transformations.

7.5.1 Current Matrix Mode (mmode)

The graphics system maintains three transformation matrices—the ModelView matrix, the Projection matrix, and the Texture matrix. As described at the beginning of this chapter, the ModelView matrix transforms coordinates from object coordinates to eye coordinates. The Projection matrix transforms coordinates from eye coordinates to clip coordinates. The Texture matrix transforms texture coordinates directly from object coordinates to clip coordinates. Its transformation is typically unrelated to that specified by the ModelView and Projection matrices.

Projection commands, such as `perspective`, `window`, and `ortho`, always replace the Projection matrix, regardless of the current matrix mode. Modeling and viewing commands, however, modify the "current" transformation matrix, as specified by the current matrix mode. When matrix mode is MVIEWING, modeling and viewing commands premultiply the ModelView matrix. When matrix mode is MPROJECTION, these commands premultiply the Projection matrix. And when `mmode` is MTEXTURE, modeling and viewing commands premultiply the Texture matrix. Modeling commands include `rot`, `translate`, and `scale`; the viewing commands are `lookat` and `polarview`.

A fourth matrix mode, MSINGLE, reconfigures the graphics system to have only a single matrix that transforms vertices directly from object coordinates to clip coordinates. This mode is obsolete, and should not be used in code you develop. It is not used in any of the examples that follow this chapter. For historical reasons, however, MSINGLE matrix mode is the default operating mode of the Graphics Library. For this reason, all programs should set matrix mode to MVIEWING, MPROJECTION, or MTEXTURE before any matrix operations are performed.

`mmode` specifies which of three matrices is the current matrix: ModelView (MVIEWING), Projection (MPROJECTION), or Texture (MTEXTURE). The current matrix is the one modified by modeling and viewing commands, and the one returned by `getmatrix`.

```
void mmode(m)
short m;
```

7.5.2 Hierarchical Drawing with the Stack Matrix

A drawing can be composed of many copies of simpler drawings, each of which can be composed of still simpler drawings, and so on. For example, if you were writing a program to draw a picture of a bicycle, you might want to have one subroutine that draws a wheel, and to call that subroutine twice to draw two wheels, appropriately translated. The wheel itself might be drawn by calling the spoke drawing subroutine 36 times, appropriately rotated. In a still more complicated drawing of many bicycles, you might like to call the bicycle drawing routine many times.

Suppose the bicycle is described in a coordinate system where the bottom bracket (the hole through which the pedal crank's axle runs) is the origin. You would draw the frame relative to this origin, but translate forward a few inches before drawing the front wheel (defined, say, relative to its axis). Then you would like to remove the forward translation to get back to the bicycle's frame of reference, and translate back to draw another instance of the wheel.

What is happening mathematically is this: suppose the modeling transformation that describes the bicycle's frame of reference is *M*, and that *S* and *T* are transformations (relative to *M*) to move forward for drawing the front wheel, and back for the back wheel respectively. You would like to draw the wheel using transformation *SM* for the front wheel and *TM* for the back wheel.

This is easily accomplished using the ModelView matrix stack. At any point in a drawing, there is a current ModelView matrix that sits at the top of the matrix stack, and is composed of all the modeling and viewing transformations composed thus far. In the bicycle example, this lumped-together transformation is called *M*. Any vertex is transformed by the top matrix, which is just what you want to do for drawing the frame.

Two subroutines, `pushmatrix`,and `popmatrix`, push and pop the ModelView matrix stack. `pushmatrix` pushes the matrix stack down and copies the current matrix to the new top. Thus, after a `pushmatrix`, there are two copies of *M* on top. `translate` (by a translation matrix *S*) leaves the stack with *SM* on top and *M* underneath. The wheel is then drawn once using the *SM* transformation. `popmatrix` eliminates the *SM* on top, leaving *M*, and another `pushmatrix` makes two copies of *M*.

Translating by *T* puts *TM* on top, so you can now draw the back wheel; after popping the matrix stack again, *M* is on top, and you can draw the rest of the frame. The code to draw the bicycle might look like:

```
... /* code to get M on top of the stack */
pushmatrix();
translate(-dist_to_back_wheel, 0.0, 0.0);
drawwheel();
popmatrix();
pushmatrix();
translate(dist_to_front_wheel, 0.0, 0.0);
drawwheel();
popmatrix();
drawframe();
```

Note that the subroutine `drawwheel` might easily push and pop matrices itself before calling the `drawspoke` subroutine.

pushmatrix

`pushmatrix` pushes down the transformation stack, duplicating the current matrix. If the transformation stack contains one matrix, *M*, after a `pushmatrix`, it will contain two copies of *M*. You can modify only the top copy. Because only the ModelView matrix is stacked, you should call `pushmatrix` only while `mmode` is MVIEWING.

```
void pushmatrix()
```

popmatrix

`popmatrix` pops the transformation stack. You should call `popmatrix` only while `mmode` is MVIEWING.

```
void popmatrix()
```

Example—Hierarchical Description

This example of a simple program uses a hierarchical description of a car. It is so simple that the car body is a rectangle and the wheels are square, and everything is 2D. Note that the positions and orientations of the nine cars are independent, and each wheel has a different rotation.

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

float carbody[][2] = {{-.1, -.05},
       {.1, -.05},{.1, .05}, {-.1, .05}};
float wheel[][2] = {{-.015, -.015},
       {.015, -.015},{.015, .015}, {-.015, .015}};

main()
{
  float xoffset, yoffset;

    keepaspect(1, 1);
    winopen("hierarchy");
    mmode(MVIEWING);
    ortho2(-1.0, 1.0, -1.0, 1.0);
    color(BLACK);
    clear();
    for (xoffset = -.5; xoffset <= .5; xoffset += .5)
        for (yoffset = -.5; yoffset <= .5; yoffset += .5) {
        pushmatrix();
        translate(xoffset, yoffset, 0.0);
        rotate(rand()&0xffff, 'z');
        drawcar();
        popmatrix();
        }
    sleep(3);
    gexit();
    exit(0);
}
```

```
drawcar()
{
    float xwheel, ywheel;

    color(RED);
    bgnpolygon();
    v2f(&carbody[0][0]);
    v2f(&carbody[1][0]);
    v2f(&carbody[2][0]);
    v2f(&carbody[3][0]);
    endpolygon();
    pushmatrix();
    translate(-.1, -.05, 0.0);
    rotate(200, 'z');
    drawwheel();
    popmatrix();
    pushmatrix();
    translate(.1, -.05, 0.0);
    rotate(400, 'z');
    drawwheel();
    popmatrix();
    pushmatrix();
    translate(.1, .05, 0.0);
    rotate(600, 'z');
    drawwheel();
    popmatrix();
    pushmatrix();
    translate(-.1, .05, 0.0);
    rotate(800, 'z');
    drawwheel();
    popmatrix();
}

drawwheel()
{
    color(GREEN);
    bgnpolygon();
    v2f(&wheel[0][0]);
    v2f(&wheel[1][0]);
    v2f(&wheel[2][0]);
    v2f(&wheel[3][0]);
    endpolygon();
}
```

Example—Spirals

This shorter and perhaps more interesting program illustrates hierarchy and more complex transformations. It is a computer model of the children's toy called "Spirograph." In the toy, one plastic gear is pinned to a piece of paper, and another gear is allowed to turn around it. You place a pen through a hole in the moving gear off center, and, as the moving gear rolls around the fixed gear, you draw interesting patterns.

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

#define PEN_TO_CENTER 0.2
#define R1 .6
#define R0 .35

drawdot()
{
  translate(PEN_TO_CENTER, 0.0, 0.0);
  bgnpoint();
  v2i(0,0);
  endpoint();
}

draw1(theta)
float theta;
{
  pushmatrix();
  rot(theta, 'z');
  translate(R1 + R0, 0.0, 0.0);
  rot(-theta*R1/R0, 'z');
  drawdot();
  popmatrix();
}
```

```
main()
{
    float theta;

    keepaspect(1, 1);
    winopen("spirograph");
    mmode(MVIEWING);
    ortho2(-2.0, 2.0, -2.0, 2.0);
    color(BLACK);
    clear();
    color(WHITE);
    for (theta = 0; theta < 3600.0; theta += .25)
        draw1(theta);
    sleep(2);
    gexit();
    exit(0);
}
```

In this example, *R0* and *R1* are the radii of the two gears, and PEN_TO_CENTER is the distance from the pen to the center of the moving gear. With a slight modification of this program, you can build a super spirograph that has three levels of gears, each moving along the next at a uniform rate. It would be difficult to build this one out of plastic.

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

#define PEN_TO_CENTER 0.2
#define R2 .8
#define R1 .6
#define R0 .35

drawdot()
{
  translate(PEN_TO_CENTER, 0.0, 0.0);
  bgnpoint();
  v2i(0,0);
  endpoint();
}
```

```
draw1(theta)
float theta;
{
  pushmatrix();
  rot(theta, 'z');
  translate(R1 + R0, 0.0, 0.0);
  rot(-theta*R1/R0, 'z');
  drawdot();
  popmatrix();
}

drawx(theta)
float theta;
{
  pushmatrix();
  rot(theta, 'z');
  translate(R2 + R1, 0.0, 0.0);
  rot(-theta*R2/R1, 'z');
  draw1(theta);
  popmatrix();
}

main()
{
    float theta;

    keepaspect(1, 1);
    winopen("spirograph");
    ortho2(-3.0, 3.0, -3.0, 3.0);
    color(BLACK);
    clear();
    color(WHITE);
    for (theta = 0; theta < 18000.0; theta += .25)
        drawx(theta);
    sleep(2);
    gexit();
    exit(0);
}
```

7.6 Viewports, Screenmasks, and Scrboxes

The *viewport* is the area of the window that displays an image. You specify it in window coordinates, where the coordinates of the pixel at the lower-left corner of the window are (0, 0). The total visible screen area varies from system to system.

viewport

`viewport` specifies, in window coordinates, the area of the window that displays an image. By default, when you open a window on the screen, its viewport is set to cover the whole window. Its arguments (*left*, *right*, *bottom*, *top*) define a rectangular area on the window by specifying the left, right, bottom, and top coordinates. The portion of eye coordinates that `window`, `ortho`, or `perspective` describes is mapped into the viewport.

```
void viewport(left, right, bottom, top)
Screencoord left, right, bottom, top;
```

Although window coordinates are continuous, not discrete, the parameters passed to viewport are integer values. Thus, the viewport is always an integer number of pixels wide and high. Pixel x,y is included in the viewport if $x \geq$ left and $x \leq$ right and $y \geq$ bottom and $y \leq$ top. Because pixel centers have integer coordinates in the continuous window coordinate space, the window area included in a viewport is exactly (left-0.5) $\leq x <$ (right+0.5), (bottom-0.5) $\leq y <$ (top+0.5).

getviewport

`getviewport` returns the current viewport. Its arguments (*left, right, bottom, top*) are the addresses of four memory locations. These are assigned the left, right, bottom, and top coordinates of the current viewport.

```
void getviewport(left, right, bottom, top)
Screencoord *left, *right, *bottom, *top;
```

scrmask

`viewport` sets both the viewport and the screenmask to the same area. (The *screenmask* is a specified rectangular area of the screen to which all drawings are clipped.) The viewport maps coordinates to the window and the screenmask specifies the portion of the window to which the geometry can be drawn. The screenmask is a setting that regards only the physical display within the window. The screenmask and viewport are usually set to the same area. `scrmask` sets only the screenmask, which must be placed entirely within the viewport.

```
void scrmask(left, right, bottom, top)
Screencoord left, right, bottom, top;
```

getscrmask

`getscrmask` returns the coordinates of the current screenmask in the arguments *left*, *right*, *bottom*, and *top*.

```
void getscrmask(left, right, bottom, top)
Screencoord *left, *right, *bottom, *top;
```

pushviewport

The system maintains a stack of viewports, and the top element in the stack is the current viewport. `pushviewport` duplicates the current viewport and screenmask and pushes them on the stack.

```
pushviewport()
```

popviewport

`popviewport` pops the stack of viewports and sets the viewport and screenmask. The viewport on top of the stack is lost.

```
void popviewport()
```

scrbox

`scrbox` is a dual of the `scrmask` capability. Rather than limiting drawing effects to a screen-aligned subregion of the viewport, it tracks the screen-aligned subregion (screen box) that has been affected. Unlike `scrmask`, which defaults to the viewport boundary if not explicitly enabled, `scrbox` must be explicitly turned on to be effective.

While enabled (mode SB_TRACK), `scrbox` maintains leftmost, rightmost, lowest, and highest window coordinates of all pixels that are scan-converted. By default, `scrbox` is reset (mode SB_RESET), forcing the leftmost and lowest screen box values to be greater than the rightmost and highest screen box values. While `scrbox` is set to mode SB_HOLD, the current boundary values are unchanged, regardless of any drawing operations.

Because `scrbox` operates on the pixels that result from the scan conversion of points, lines, polygons, and characters, it correctly handles wide lines, antialiased (smooth) points and lines, and characters.

`scrbox` results are only guaranteed to bound the modified frame buffer region, but they might exceed the bounds of this region due to implementation.

```
void scrbox(arg)
long arg;
```

getscrbox

`getscrbox` returns to current `scrbox` limits in the variables *left*, *right*, *bottom*, and *top*.

```
void getscrbox(left, right, bottom, top)
long *left, *right, *bottom, *top;
```

7.7 Additional Clipping Planes

Geometry is always clipped against the boundaries of the six-plane frustum defined by the current Projection matrix. `clipplane` allows the specification of additional planes, not necessarily perpendicular to the *x*, *y*, or *z* axes, against which all geometry is clipped. You can specify up to six additional planes. Because the resulting clipping region is always the intersection of the (up to) 12 half-spaces, it is always convex.

`clipplane` uses the following format:

```
void clipplane(index,mode,params)
long index,mode;
float params[];
```

where its arguments have the following meanings:

index Expects an integer in the range 0 through 5. Indicates which of the six clipping planes is being modified.

mode Expects one of three tokens:

- CP_DEFINE — Use the plane equation passed in *params* to define a clipping plane. The clipping plane is neither enabled nor disabled.
- CP_ON — Enable the (previously defined) clipping plane.
- CP_OFF — Disable the clipping plane. (default)

params Expects an array of four floats that specify a plane equation. A plane equation is usually thought of as a four-vector *A,B,C,D*. In this case, *A* is the first component of the *params* array, and *D* is the last. A four-component vertex array (see *v4f*) can be passed as a plane equation, where vertex *X* becomes *A*, *Y* becomes *B*, etc.

`clipplane` specifies a half-space using a four-component plane equation. When it is called with mode CP_DEFINE, this object coordinate plane equation is transformed to eye coordinates using the inverse of the current ModelView matrix. (You cannot call or use `clipplane` when the matrix mode is MSINGLE.)

$$P_{object} = \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix}$$

$$P_{eye} = M^{-1}_{modelview} * P_{object}$$

Once you have defined a clipping plane, you enable it by calling `clipplane` with the CP_ON argument, and with arbitrary values passed in *params*. While the program is drawing after a clipping plane has been defined and enabled, each vertex is transformed to eye coordinates, where it is dotted with the clipping plane P_{eye}. Eye coordinates whose dot product with P_{eye} is positive or zero are in, and require no clipping. Those eye coordinate vertices whose dot product is negative are clipped. Because `clipplane` clipping is done in eye coordinates, changes to the Projection matrix have no effect on its operation.

By default, all six clipping planes are undefined and disabled. The behavior of an enabled but undefined clipping plane is also undefined.

It is sometimes convenient to define a clipping plane based on a point, and a direction in object coordinates. A point and a normal are converted to a plane equation by the following arithmetic:

$$\text{point} = [Px, Py, Pz]$$

$$\text{normal} = \begin{bmatrix} Nx \\ Ny \\ Nz \end{bmatrix}$$

$$\text{plane equation} = \begin{bmatrix} A \\ B \\ C \\ D \end{bmatrix} = \begin{bmatrix} Nx \\ Ny \\ Nz \\ -[Px, Py, Pz] \cdot \begin{bmatrix} Nx \\ Ny \\ Nz \end{bmatrix} \end{bmatrix}$$

7.8 User-Defined Transformations

Modeling and viewing transformation commands premultiply the current matrix (one of ModelView, Projection, or Texture) with a 4 x 4 matrix that they compute based on their parameters. You can also premultiply, or replace, the current matrix with a 4 x 4 matrix of your own description.

multmatrix

`multmatrix` premultiplies the current matrix by the given matrix. That is, if T is the current matrix, `multmatrix(M)` replaces T with MT. The current matrix is the ModelView matrix on the top of the ModelView stack (if `mmode` is MVIEWING), the Projection matrix (if `mmode` is MPROJECTION), or the Texture matrix (if `mmode` is MTEXTURE).

```
multmatrix(m)
Matrix m;
```

getmatrix

`getmatrix` copies the current matrix to an array provided by the user. The current matrix is the ModelView matrix on the top of the ModelView stack (if `mmode` is MVIEWING), the Projection matrix (if `mmode` is MPROJECTION), or the Texture matrix (if `mmode` is MTEXTURE).

```
getmatrix(m)
Matrix m;
```

loadmatrix

loadmatrix loads a 4x4 floating point matrix onto the stack, replacing the current top of the stack.

```
loadmatrix(m)
Matrix m;
```

`loadmatrix` replaces the current matrix with matrix `m`. The current matrix is the ModelView matrix on the top of the ModelView stack (if `mmode` is MVIEWING), the Projection matrix (if `mmode` is MPROJECTION), or the Texture matrix (if `mmode` is MTEXTURE).

8. Hidden Surface Removal

By default, IRIS-4D Series systems do no hidden surface removal—figures are rendered on the screen in the order they are drawn. For most 3-D drawings, it is important to draw only those surfaces that are nearest to the eye, at least for opaque objects; all other surfaces would be obscured by those nearer the eye. The time it takes the system to draw surfaces that will not be visible in the final scene can cause a noticeable degradation of performance in complex scenes where drawing speed is important.

There are many ways to do hidden surface removal, depending on the types of figures being rendered. The primary method used on IRIS-4D Series systems is a *z*-buffer, which calculates the distance to the eye from the surfaces covering each pixel, and draws only the surface that is the closest. The calculation has to be done on a per-pixel basis, because it is possible to have a set consisting of as few as three polygons, each of which is overlapped by another in the set (see Figure 8-1).

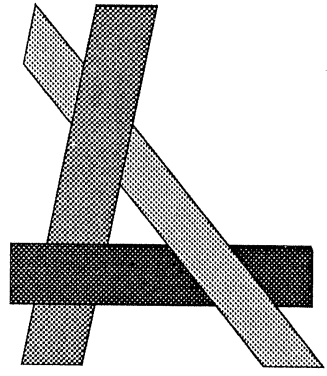


Figure 8-1. Overlapping Polygons

Another hidden surface removal method supported by IRIS-4D Series systems is backfacing polygon removal. For many objects including all convex polygonal 3-D figures, if each polygonal face is drawn in counterclockwise order when viewed from outside the object, then after transformation, the faces on the front are in counterclockwise order and those on the back are in clockwise order. The system can be put in a mode where only counterclockwise polygons are drawn, and for cases such as this, the 3-D figures are correctly rendered with hidden surfaces removed.

Backfacing polygon removal is not as general as *z*-buffering, but *z*-buffering is much slower than backface removal on earlier systems. `backface` and `frontface` are also useful for operations other than hidden surface removal.

8.1 *z*-buffering

The *z*-buffer is a set of 24-bit integers, one associated with each pixel on the screen. To use it, turn on *z*-buffering mode and set all the numbers to the largest possible positive value. Then as each polygon, line, point, or character is rendered, its *x* and *y* screen coordinates are calculated in the usual way; however, before the pixel is set to the polygon's color, the *z* coordinate is also calculated. The *z* coordinate is effectively the distance to the eye. This *z* value is compared to the *z*-buffer value for that pixel. If the *z* value is smaller than the value in the *z*-buffer, the pixel is colored, and the pixel's *z*-buffer value is set to the new *z* value. Thus at any point in the drawing, the values in the *z*-buffer represent the distance to the item that is currently closest to the eye. The color value stored in the bitplanes represents the color of that item. The *z* comparison is signed on IRIS-4D/B/G/VGX systems and on Personal IRIS systems; it is unsigned on IRIS-4D/GT/GTX systems.

Another consideration when using *z*-buffering is that the *near* and *far* values in the call to `perspective` have a profound effect on the resolution of the *z*-buffer's comparison facility. Because the *z*-buffer contains a fixed and finite number of integer values that can be used to compare against the *z* value of the object in the scene, you control the resolution of the *z*-buffer by setting the near and far values. The more closely you can bracket the objects between the near and far clipping planes, the better *z*-compare resolution you achieve. It is particularly important to move the near clipping plane as far from the viewer position as possible.

The program illustrates *z*-buffering:

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

float v0[3] = {-1.0, -1.0, -1.0};
float v1[3] = {-1.0, -1.0, 1.0};
float v2[3] = {-1.0, 1.0, 1.0};
float v3[3] = {-1.0, 1.0, -1.0};
float v4[3] = {1.0, -1.0, -1.0};
float v5[3] = {1.0, -1.0, 1.0};
float v6[3] = {1.0, 1.0, 1.0};
float v7[3] = {1.0, 1.0, -1.0};

long delaycount;

main(argc, argv)
int    argc;
char   *argv[];
{
    long xrot, yrot, zrot;

    xrot = yrot = zrot = 0;
    if (argc == 1)
    delaycount = 0;
    else
    delaycount = 1;
    keepaspect(1, 1);
    winopen(argv[0]);
    RGBmode();
    if (delaycount == 0)
    doublebuffer();
    gconfig();
    mmode(MVIEWING);
    ortho(-4.0, 4.0, -4.0, 4.0, -4.0, 4.0);
    while (1) {
    pushmatrix();
    rotate(xrot, 'x');
    rotate(yrot, 'y');
    rotate(zrot, 'z');
    xrot += 11;
    yrot += 15;
```

```
if (xrot + yrot > 3500) zrot += 23;
    if (xrot > 3600) xrot -= 3600;
    if (yrot > 3600) yrot -= 3600;
    if (zrot > 3600) zrot -= 3600;
    cpack(0);
    clear();
    if (getbutton(LEFTMOUSE)) {
        zbuffer(TRUE);
        zclear();
    } else
        zbuffer(FALSE);
    pushmatrix();
    scale(1.2, 1.2, 1.2);
    translate(.3, .2, .2);
    drawcube();
    popmatrix();
    pushmatrix();
    rotate(450+zrot, 'x');
    rotate(300-xrot, 'y');
    scale(1.8, .8, .8);
    drawcube();
    popmatrix();
    pushmatrix();
    rotate(500+yrot, 'z');
    rotate(-zrot, 'x');
    translate(-.3, -.2, .6);
    scale(1.4, 1.2, .7);
    drawcube();
    popmatrix();
    if (delaycount == 0)
        swapbuffers();
    popmatrix();
     }
}

delay()
{
    sleep(delaycount);
}
```

```
drawcube()
{
    cpack(0xff);
    bgnpolygon();
    v3f(v0); v3f(v1); v3f(v2); v3f(v3);
    endpolygon();
    delay();
    cpack(0xff00);
    bgnpolygon();
    v3f(v0); v3f(v4); v3f(v5); v3f(v1);
    endpolygon();
    delay();
    cpack(0xff0000);
    bgnpolygon();
    v3f(v4); v3f(v7); v3f(v6); v3f(v5);
    endpolygon();
    delay();
    cpack(0xffff);
    bgnpolygon();
    v3f(v3); v3f(v7); v3f(v6); v3f(v2);
    endpolygon();
    delay();
    cpack(0xff00ff);
    bgnpolygon();
    v3f(v5); v3f(v1); v3f(v2); v3f(v6);
    endpolygon();
    delay();
    cpack(0xffff00);
    bgnpolygon();
    v3f(v0); v3f(v4); v3f(v7); v3f(v3);
    endpolygon();
}
```

The program draws three cubical objects (they are all originally perfect cubes, but `scale` stretches them along their axes). The objects tumble through each other and the whole scene is also rotating. While the left mouse button is up, the scene is drawn without *z*-buffering; when you press it, *z*-buffering is enabled. If the program is called with an argument, there is a short delay between drawing each polygon. In this mode, the left mouse button still controls the *z*-buffering.

The key part of the program that turns on the *z*-buffering is the pair of subroutines:

```
zbuffer(TRUE);
zclear();
```

The first routine enables *z*-buffer comparisons to be made before each write, and the second sets all the *z* values to the largest possible value for pixels in the viewport. In this example, `zbuffer(TRUE)` is called for every frame; however, this is not necessary because a typical program turns it on at the beginning. The code is written as it is because the left mouse button can come up at any time, in which case *z*-buffering should be turned off.

`getzbuffer` returns TRUE or FALSE depending on whether *z*-buffering is enabled or not. By default, *z*-buffering is turned off.

8.2 Controlling *z* values

Just as `viewport` controls the scaling of the *x* and *y* coordinates, there is a subroutine, `lsetdepth`, that controls the scaling of the z coordinates. It takes two arguments of type `long`, corresponding to the near and far planes. By default, near is set to the minimum value that can be stored in the *z*-buffer and far is set to the maximum value. These values are system-dependent (see Table 8-1),

System Model	Minimum Value	Maximum Value
B or G	0x4000	0x3FFF
GT or GTX	0	0x7FFFFF
Personal IRIS or VGX	0x800000	0x7FFFFF

Table 8-1. lsetdepth Values for IRIS-4D Series Systems

czclear

A common code sequence in programs that do *z*-buffering is:

```
color(0);
clear();
zclear();
```

This code clears the color bitplanes to zero and clears the *z*-buffer bitplanes to the maximum value. Unfortunately, it takes a relatively long time, because `clear` touches each pixel, then `zclear` touches each pixel. In some hardware implementations (for example, IRIS-4D/GT/GTX systems), the hardware can in certain cases simultaneously clear the color planes and the *z*-buffer planes. `czclear` does this.

```
czclear(color, zval)
long color, zval
```

`czclear` clears the bitplanes to *color* and the *z*-buffer to *zval* simultaneously.

IRIS-4D GT and GTX systems can do a simultaneous clear only under the following circumstances:

- In RGB mode, the 24 least significant bits of color (red, green, and blue) must be identical to the 24 least significant bits of *zval*. In the case of RGB mode, it is common to set the background color to black (all zeros). This makes it necessary for you, in effect, to reverse the orientation of the z-buffer near/far clipping values. The following two function calls reverse the z-buffer orientation so that the maximum distance to which you can initially clear all z values is 0 instead of 0x7fffff :

```
lsetdepth(0x7fffff, 0x0);
zfunction(ZF_GEQUAL);
```

  At this point, all that has changed is that the system has positioned the viewer so that all z compares take place with *near* mapped to a large number and *far* mapped to 0.

- In color map mode, the 12 least significant bits of color must be identical to the 12 least significant bits of *zval*. Because the color parameter is expected to be an index into the color map in color map mode, only the lowest 12 bits are significant.

On IRIS-4D/GT/GTX systems, use `czclear` to clear both the z-buffer and the bitplanes to the same values at the same time. A simultaneous clear happens if circumstances allow it.

On the Personal IRIS, you can speed up `czclear` by as much as a factor of four for common values of *zval* if you call `zfunction` with it, so that one of the conditions in Table 8-2 is met:

zval	zfunction
0x800000	ZF_GREATER or ZF_GEQUAL
0x7fffff	ZF_LESS or ZF_LEQUAL

Table 8-2. zfunction Values for Personal IRIS

8.3 Special features

This section discussed special features that control *z*-buffering. Most of them are rarely used, so this section can be skipped on first reading. Topics include: writing directly into the *z*-buffer, using alternate depth comparison functions and sources, and using writemasks for the *z*-buffer.

8.3.1 Drawing into the *z*-Buffer

There are certain applications where it is useful to write values directly into the *z* buffer.

`zclear` is actually a special case of this, where the values in the *z*-buffer are all set to some depth value. In a flight simulator, for example, suppose that the view on the screen includes an instrument panel wrapped around a window. If the instrument panel does not change from frame to frame, there is no reason to redraw it, so it might be nice to clear only the portion of the screen and *z*-buffer corresponding to the plane's window and simply redraw the outside scene for each frame.

To do this, set the "color" to the value returned by the call to `getgdesc(GD_ZMAX)` on your system, use `zdraw` to draw these values into the *z*-buffer, and draw the polygon(s) representing the window. When the outside view is drawn, it is always masked by the plane's window frame and instrument panel (which should be closer to the eye). Thus an extremely complex instrument panel is possible, since it needs to be drawn only once.

The following sample program illustrates the above technique using a spinning cube as the outside scene, and an array of 25 circles as the window.

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

float vert0[3] = {-1.0, -1.0, -1.0};
float vert1[3] = {-1.0, -1.0, 1.0};
float vert2[3] = {-1.0, 1.0, 1.0};
float vert3[3] = {-1.0, 1.0, -1.0};
float vert4[3] = {1.0, -1.0, -1.0};
float vert5[3] = {1.0, -1.0, 1.0};
float vert6[3] = {1.0, 1.0, 1.0};
float vert7[3] = {1.0, 1.0, -1.0};
```

```
drawcube()
{
    cpack(0xff0000);
    bgnpolygon();
    v3f(vert0); v3f(vert1); v3f(vert2); v3f(vert3);
    endpolygon();

    cpack(0xff00);
    bgnpolygon();
    v3f(vert3); v3f(vert2); v3f(vert6); v3f(vert7);
    endpolygon();

    cpack(0xffff);
    bgnpolygon();
    v3f(vert7); v3f(vert6); v3f(vert5); v3f(vert4);
    endpolygon();

    cpack(0xffff00);
    bgnpolygon();
    v3f(vert4); v3f(vert5); v3f(vert1); v3f(vert0);
    endpolygon();

    cpack(0xff00ff);
    bgnpolygon();
    v3f(vert1); v3f(vert2); v3f(vert6); v3f(vert5);
    endpolygon();

    cpack(0xffffff);
    bgnpolygon();
    v3f(vert0); v3f(vert4); v3f(vert7); v3f(vert3);
    endpolygon();
}

main()
{
    long i, j;
    float x, y;
```

```
    keepaspect(1, 1);
    winopen("zdraw");
    RGBmode();
    doublebuffer();
    gconfig();
    mmode(MVIEWING);
    ortho(-2.0, 2.0, -2.0, 2.0, -2.0, 2.0);
    zbuffer(TRUE);
    zclear();
    cpack(0xff);
    pushmatrix();
    translate(0.0, 0.0, 1.9);
    frontbuffer(TRUE);
    rectf(-3.0, -3.0, 3.0, 3.0);
    frontbuffer(FALSE);
    popmatrix();
    while(1) {
      cpack(0x7fffff);
     zbuffer(FALSE);
     zdraw(TRUE);
     for (x = -1.0; x <= 1.0; x += .5)
        for (y = -1.0; y <= 1.0; y += .5)
       circf(x, y, 0.2);
     zdraw(FALSE);
     zbuffer(TRUE);
     pushmatrix();
     translate(0.0, 0.0, -1.9);
     cpack(0x00);
     rectf(-3.0, -3.0, 3.0, 3.0);
     popmatrix();
     pushmatrix();
     rotate(i, 'x');
     rotate(j, 'y');
     i += 11; j += 17;
     if (i > 3600) i -= 3600;
     if (j > 3600) j -= 3600;
     drawcube();
     popmatrix();
     swapbuffers();
     }
}
```

This program is written in RGB mode, because in color map mode, every color, including the "color" drawn into the z-buffer, is masked to 12 bits. When `zdraw` is TRUE, `zbuffer` must be set to FALSE; otherwise, both would try to alter the z-buffer contents simultaneously.

The idea behind the program is that a red plane is drawn nearer the eye than anything else drawn. This sets all the z-buffer values so they record data near the eye. In the main loop, 25 holes are drilled into the red plane by setting the z-buffer values in the circles to indicate that the surface is far away. Then a black background is drawn farther from the eye than any part of the cube. Finally, the cube is drawn. It is visible only through the 25 holes.

`zdraw` is similar to `frontbuffer` and `backbuffer` in that it permits writing into the z-buffer bank. Normally, if you are writing into the z-buffer, you do not want to write into the front buffer or back buffer at the same time. In the example above, this does not matter, since the first subroutine called after the z-buffer is written to clears to black. Usually, drawing into the z-buffer should be bracketed by subroutines that set `backbuffer(FALSE)` and then `backbuffer(TRUE)` afterward (assuming the program is in double buffer mode).

In single buffer mode, `frontbuffer` normally has no effect. However, if you call `frontbuffer(FALSE)`, a flag is set so that when `zdraw` is TRUE, the front buffer (the only buffer in single buffer mode) is not written. If `zdraw` is FALSE, `frontbuffer(FALSE)` has no effect.

8.3.2 Alternative Comparisons

In the default mode, the *z* coordinate of the incoming pixel is compared to the *z* coordinate of the geometry already drawn at that pixel. If the incoming *z* value shows that the new geometry is closer to the eye than the old one, the values of the old pixel and of the old *z* value are replaced by the new ones.

Thus the new value is compared to the old, and if it is less than the old, the old quantities are replaced. It is possible to change the comparison function from less-than to many other things. The available comparisons are:

ZF_NEVER	Never overwrite the source pixel value.
ZF_LESS	Overwrite the source pixel value if the *z* of the source pixel value is less than the *z* of the destination value.
ZF_EQUAL	Overwrite the source pixel value if the *z* of the source pixel value is equal to the *z* of the destination value.
ZF_LEQUAL	(default) Overwrite the source pixel value if the *z* of the source pixel value is less than or equal to the *z* of the destination value.
ZF_GREATER	Overwrite the source pixel value if the *z* of the source pixel value is greater than the *z* of the destination value.
ZF_NOTEQUAL	Overwrite the source pixel value if the *z* of the source pixel value is not equal to the *z* of the destination value.
ZF_GEQUAL	Overwrite the source pixel value if the *z* of the source pixel value is greater than or equal to the *z* of the destination value.
ZF_ALWAYS	Always overwrite the source pixel value regardless of destination value.

Using various comparisons, it is possible to use the *z*-buffer for various things. To change the comparison function, use `zfunction`, a subroutine that takes a single argument chosen from among the eight listed above.

It is also possible to do comparisons on color buffers rather than on the z-buffer. This is useful primarily for drawing antialiased lines that cross each other (see Chapter 15). The function that changes the source buffer for z-buffer type comparisons is called `zsource`. It has a single argument that can be either ZSRC_DEPTH (the default) or ZSRC_COLOR.

8.3.3 *z*-buffer Writemasks

Finally, `zwritemask` can be used like other writemasks to control writing into the z-buffer. It might be useful for a very complicated background into which a few objects are going to be drawn and moved quickly. Setting `zwritemask` to zero locks the background information in, and prevents its modification, but the new objects are drawn or not depending on the depth comparison.

8.3.4 Stenciling on IRIS-4D/VGX Systems

IRIS-4D/VGX systems support an additional z-buffer, like test, that uses a different algorithm from the one described previously in this chapter. This test uses the VGX's flexible frame buffer configuration and lets you allocate *stencil planes* to be used for this test.

Stenciling, like z-buffering, enables and disables drawing on a per-pixel basis. You draw into the stencil planes using GL drawing primitives, and then render geometry and images, using the stencil planes to mask out portions of the screen. Stenciling is typically used in multi-pass rendering algorithms to achieve special effects, like decals, outlining, and constructive solid geometry rendering.

stensize

`stensize` lets you define the bitplanes that you wish to use as the stencil. You can define up to 8 stencil planes. IRIS-4D/VGX systems without the optional alpha bitplanes allocate the stencil bitplanes from the least significant planes of the *z*-buffer. Use `getgdesc` to determine whether your system has alpha bitplanes.

Once you have allocated some number of bitplanes for use as a stencil, these planes can be used to store information that is later used by the `stencil` statement.

```
void stensize(planes)
long planes;
```

sclear

sclear sets the value of every pixel in the currently allocated stencil buffer. You pass the desired value to `sclear` as *sval*. The clearing operation is limited by the current `viewport` and `scrmask` statements in effect, and is masked by the current `swritemask`.

```
void sclear(sval)
unsigned long sval;
```

swritemask

`swritemask` lets you specify which of the stencil bitplanes can be modified by `sclear` and normal stencil operation.

```
void swritemask(mask)
unsigned long mask;
```

stencil

The `stencil` statement controls testing of stencil bitplanes before the system writes to the frame buffer. The first argument to `stencil` enables or disables it. When stenciling is enabled, the system tests the defined stencil bitplanes for each pixel against a programmed reference value before writing to that pixel. Based on the contents of the stencil bitplanes and the programmed tests defined by the `stencil` statement, the system then conditionally modifies the pixel's contents (both for the color bitplanes and also for the z-buffer) by some programmed value, as defined in the call to `stencil`.

To enable stenciling, call `stencil` with *enable* set to TRUE. If you call `stencil` with *enable* set to FALSE, the following parameters are ignored and stencil testing is not performed.

```
void stencil(enable, ref, func, mask, fail, pass, zpass)
long enable;
unsigned long ref;
long func;
unsigned long mask;
long fail, pass, zpass;
```

Once you enable stenciling, the system tests the color and z-buffer bitplanes for each pixel. The results of the tests are determined by a reference value, passed through the argument *ref*, the value in the stencil bitplanes, and a stencil function that operates on them both. This function, passed as the argument *func*, can be one of the following:

SF_NEVER — Do not perform the specified stencil update (passed as the value of *pass*, *fail*, and *zpass*) regardless of the results of the comparison.

SF_LESS — Perform the specified stencil update (specified as the value of *pass*, *fail*, and *zpass*) if *ref* is less than the value in the stencil planes.

SF_EQUAL — Perform the specified stencil update (specified as the value of *pass*, *fail*, and *zpass*) if *ref* is equal to the value in the stencil planes.

SF_LEQUAL Perform the specified stencil update (specified as the value of *pass*, *fail*, and *zpass*) if *ref* is less than or equal to the value in the stencil planes.

SF_GREATER Perform the specified stencil update (specified as the value of *pass*, *fail*, and *zpass*) if *ref* is greater than the value in the stencil planes.

SF_NOTEQUAL Perform the specified stencil update (specified as the value of *pass*, *fail*, and *zpass*) if *ref* is not equal to the value in the stencil planes.

SF_GEQUAL Perform the specified stencil update (specified as the value of *pass*, *fail*, and *zpass*) if *ref* is greater than or equal to the value in the stencil planes.

SF_ALWAYS Perform the specified stencil update (specified as the value of *pass*, *fail*, and *zpass*) regardless of the results of the comparison.

The `mask` argument to stencil defines which stencil bitplanes are significant during the comparison operation. Use this argument to ignore individual planes you do not want to use in the stencil test.

When stencil performs its test as defined by *func*, it returns one of three possible values :

fail The stencil test (defined in the call to stencil) fails

pass The stencil test passes (but the z-buffer test fails

zpass The stencil test passes, and the z-buffer test passes

(If the z-buffer is not enabled, only *fail* and *pass* are considered.)

These three possible values are reflected as the arguments *fail*, *pass*, and *zpass* (the last three arguments passed to stencil). These arguments define the operation to be performed based on the results of the stencil test. The system performs one of the functions defined by the value of *fail*, *pass*, and *zpass* as passed to `stencil`.

You must pass one of the following flags as the arguments *fail*, *pass*, and *zpass*:

ST_KEEP	Keep the value currently in the bitplanes (no change).
ST_ZERO	Replace the contents of the pixel with zeros.
ST_REPLACE	Replace the contents of the pixel with the value of *ref*.
ST_INCR	Add 1 to the contents of the pixel. This is clamped to the maximum value of the pixel at that location.
ST_DECR	Subtract 1 from the contents of the pixel. This is clamped to 0.
ST_INVERT	Invert all bits in that pixel.

Based on the results of the test, the system performs the function that applies to the conditions. For example, in the following case:

```
stencil(TRUE, 220, SF_EQUAL, 0xff, ST_REPLACE,
        ST_KEEP, ST_KEEP);
```

the system compares 220 against the contents of the stencil planes. (Because *mask* is 0xff, all eight planes are valid in this comparison.) The test is to see whether the stencil planes are exactly equal to *ref*, which is 220. If the test fails — that is, if the contents of the stencil planes do not equal *ref* — the system replaces them with the value of *ref* (220). Both *pass* and *zpass* are set to ST_KEEP, which means that there is no change to the pixels or to the *z*-buffer if the test passes. If the *z*-buffer is enabled, color and depth are drawn only in the *zpass* case (meaning that both color and *z*-buffer planes pass the test). If the *z*-buffer is not enabled, *zpass* is ignored and only the *pass* function is performed.

The following program is an example that uses the stenciling capability to render the outline of an object in an arbitrary image. Because the bitmap of an image takes a lot of space, the example code mimics drawing the image by actually drawing polygons.

```
void backface(b)
Boolean b;

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

float rect0[4][2] = {
    {-0.5, -0.5},
    { 0.5, -0.5},
    { 0.5,  0.5},
    {-0.5,  0.5},
    };

main()
{
     if (getgdesc(GD_BITS_STENCIL) == 0) {
    fprintf(stderr, "stencil not available on this
machine\n");
    return 1;
     }

     prefsize(400, 400);
     winopen("stencil");
     RGBmode();
     stensize(3);
     gconfig();
     mmode(MVIEWING);
     ortho2(-10.0, 10.0, -10.0, 10.0);

     cpack(0);
     clear();
     sclear(0);
```

```
    wmpack(0);
    stencil(1, 0x0, SF_ALWAYS, 0x7, ST_INCR, ST_INCR,
ST_INCR);
    viewport(0, 398, 0, 398);
    checker();
    viewport(1, 399, 0, 398);
    checker();
    viewport(1, 399, 1, 399);
    checker();
    viewport(0, 398, 1, 399);
    checker();

    stencil(1, 0x0, SF_NOTEQUAL, 0x3, ST_KEEP, ST_KEEP,
ST_KEEP);
    wmpack(0xffffffff);
    scale(15.0, 15.0, 0.0);
    bgnpolygon();
   v2f(rect0[0]);
   v2f(rect0[1]);
   v2f(rect0[2]);
   v2f(rect0[3]);
    endpolygon();

    sleep(10);
    gexit();
    return 0;
}
```

```
checker()
{
    int   i,j;

    cpack(0x0000ff00);
    pushmatrix();
    translate(-5.0, -5.0, 0.0);
    for (i=0; i<9; i++) {
   for (j=0; j<9; j++) {
        translate(1.0, 0.0, 0.0);
        if ((i^j) & 0x1) {     /* checker board */
       bgnpolygon();
            v2f(rect0[0]);
            v2f(rect0[1]);
            v2f(rect0[2]);
            v2f(rect0[3]);
             endpolygon();
        }
   }
   translate(-9.0, 1.0, 0.0);
    }
    popmatrix();
}
```

The image is a basic checkerboard on black background. It "jitters" four times and increments the stencil value each time a pixel is hit. This leaves the outlines with stencil values of 0x1, 0x2, and 0x3, while pixels with stencil values of 0x0 and 0x4 are completely outside and inside the object, respectively. The last step is to render a polygon over the entire region, turning on only those pixels that are on the outline.

8.4 Eliminating Backfacing Polygons

In a scene composed entirely of opaque surfaces, backwards-facing (or backfacing) polygons are never visible. Eliminating these invisible polygons from the scene has an obvious benefit — it speeds drawing time by not drawing some of the polygons in the scene. In this way, eliminating backfacing polygons is done by a subset of hidden-surface removal.

A backfacing polygon is defined as a polygon whose vertices appear in clockwise order in screen space. When backfacing polygon removal is turned on, only polygons whose vertices appear in counterclockwise order are displayed, that is, polygons that point toward you. Therefore, the vertices of all polygons should be specified in counterclockwise order.

The idea is that if the polygons making up a surface are all oriented the same way, and if the surface is convex, after transformation, all the polygons on the front have one orientation and all those on the back have the opposite orientation. A special mode can be turned on to check whether the transformed polygons were to be drawn clockwise or counterclockwise, and only those oriented counterclockwise are drawn. The method does not always work if the object being drawn is not convex, or if there is more than one object.

backface

`backface` initiates or terminates backfacing polygon removal. The backface utility is used to improve the performance of programs that represent solid shapes as collections of polygons. The vertices of the polygons on the "far" side of the solid are in clockwise order and are not drawn.

`backface` takes a single argument. TRUE enables backface polygon elimination, and FALSE (the default) disables it.

```
void backface(b)
Boolean b;
```

getbackface

`getbackface` returns the state of backfacing filled-polygon removal. If backface removal is on, the system draws only those polygons that face the viewer. If backfacing polygon removal is enabled, 1 is returned; otherwise 0 is returned.

```
long getbackface()
```

8.5 Alpha Comparison

On IRIS-4D/VGX systems, you can also use the alpha planes to determine whether to draw pixels by comparing incoming alpha values to a reference constant value.

The `afunction` statement compares source alpha values against a reference value that you include in the `afunction` call. You also specify a comparison function, which determines the conditions under which `afunction` permits the system to draw pixels.

```
void afunction(ref, func)
long ref, func;
```

`afunction` compares alpha values of source pixels to the value of *ref*. Then, depending on the value of *func*, `afunction` determines whether a pixel is completely transparent and draws the pixel conditional to its transparency. `afunction` assumes that alpha values are proportional to pixel coverage, which is the case if you are using `pointsmooth` and `linesmooth`.

To make the system avoid drawing invisible pixels, call `afunction` as follows:

```
afunction(0, AF_NOTEQUAL);
```

This call makes the system draw pixels only if their alpha value is not equal to 0. Pixels with 0 alpha are presumed to be completely transparent (according to the conventions of `pointsmooth` and `linesmooth`).

To return the system to its default operation, call `afunction` as follows:

```
afunction(0, AF_ALWAYS);
```

This call causes the alpha hardware to compare the values of all pixels in the normal manner.

The following example uses `afunction` to define the shape of a building and its windows. (See Chapter 18 to understand how to use the texturing capability.)

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

float mt0[3][3] = {     /* mountain 0 coordinates */
    {-15.0, -10.0, -15.0},
    { 10.0, -10.0, -15.0},
    { -5.0,   5.0, -15.0},
};

float mt1[3][3] = {     /* mountain 1 coordinates */
    {-10.0, -10.0, -17.0},
    { 15.0, -10.0, -17.0},
    {  6.0,  12.0, -17.0},
};

float bldg[4][3] = {        /* building coordinates */
    {-8.0, -10.0, -12.0},
    { 8.0, -10.0, -12.0},
    { 8.0,  10.0, -12.0},
    {-8.0,  10.0, -12.0},
};

float tbldg[4][2] = {       /* building texture coordinates
*/
    {0.0, 1.0},
    {1.0, 1.0},
    {1.0, 0.0},
    {0.0, 0.0},
};
```

```
/*
 * Building texture and texture environment
 */

unsigned long    bldgtex[8*4] = {
    0xffffffff, 0xffff0000, 0x00000000, 0x00000000,
    0xffffffff, 0xffff0000, 0x0000ffcf, 0xffcfffcf,
    0xffff0000, 0xffff0000, 0x0000ffcf, 0x0000ffcf,
    0xffffffff, 0xffffffff, 0xffffffcf, 0xffcfffcf,
    0xffff0000, 0xffffffff, 0xffffffcf, 0x0000ffcf,
    0xffffffff, 0xffffffff, 0xffffffcf, 0xffcfffcf,
    0xffff0000, 0xffff0000, 0x0000ffcf, 0x0000ffcf,
    0xffffffff, 0xffff0000, 0x0000ffcf, 0xffcfffcf,
};

float txlist[] = {TX_MAGFILTER, TX_POINT, TX_NULL};
float tvlist[] = {TV_NULL};

main()
{
    if (getgdesc(GD_BITS_NORM_ZBUFFER) == 0) {
    fprintf(stderr, "Z-buffer not available on this
machine\n");
    return 1;
    }
    if (getgdesc(GD_TEXTURE) == 0) {
    fprintf(stderr, "Texture mapping not available on this
machine\n");
    return 1;
    }
    if (getgdesc(GD_AFUNCTION) == 0) {
    fprintf(stderr, "afunction not available on this
machine\n");
    return 1;
    }

    prefsize(400, 400);
    winopen("afunction");
    RGBmode();
    gconfig();
    mmode(MVIEWING);
    ortho(-20.0, 20.0, -20.0, 20.0, 10.0, 20.0);
    zbuffer(TRUE);
    czclear(0, getgdesc(GD_ZMAX));
```

```
    /*
     * Draw 2 mountains
     */
    cpack(0xff3f703f);
    bgnpolygon();
    v3f(mt0[0]);
    v3f(mt0[1]);
    v3f(mt0[2]);
    endpolygon();

    cpack(0xff234f00);
    bgnpolygon();
    v3f(mt1[0]);
    v3f(mt1[1]);
    v3f(mt1[2]);
    endpolygon();

    /*
     * Draw the building
     */
    texdef2d(1, 2, 8, 8, bldgtex, 0, txlist);
    tevdef(1, 0, tvlist);
    texbind(TX_TEXTURE_0, 1);
    tevbind(TV_ENV0, 1);
    afunction(0, AF_NOTEQUAL);
    cpack(0xffffffff);
    bgnpolygon();
    t2f(tbldg[0]);
    v3f(bldg[0]);
    t2f(tbldg[1]);
    v3f(bldg[1]);
    t2f(tbldg[2]);
    v3f(bldg[2]);
    t2f(tbldg[3]);
    v3f(bldg[3]);
    endpolygon();
    sleep(10);
    gexit();
    return 0;
}
```

9. Lighting

This chapter describes the Graphics Library lighting facility and shows how you can use this facility to create lighted objects in 3-D.

9.1 What is GL Lighting?

The interaction between light and objects is far more complicated than can be simulated in real time. Lighting on the IRIS-4D achieves a balance between realistic appearance and real-time drawing speed.

The GL interface to lighting lets you control the lights and the materials with which you draw polygons and other primitives.

In the real world, when light falls on an opaque object, some of this light is absorbed by the object and the rest of the light is reflected. Our eyes use this reflected light to interpret the shape, color, and other details about the object. The light from the illumination source is called the incident light; that which reflects off the object's surface is called reflected light.

The interaction between the incident light and the surface material creates the reflected light. You can control the way lighting creates images by manipulating the characteristics of the incident light and the surface of the objects.

9.1.1 Light Source and Surface Characteristics

The Graphics Library lighting model determines how the incident light is modified when it reflects from objects in the scene. For example, an object can appear blue for either of the following reasons:

- a white light source shines on the object and it reflects only the wavelengths that our eyes interpret as blue. Although other wavelengths of light are present, the object absorbs them and reflects only blue.
- the object reflects blue light and possibly other wavelengths, but only blue light is shining on it. In this case, there are no other wavelengths for the object to reflect.

In the lighting model, as in the real world, the characteristics of the light source determine the direction, intensity, and wavelength with which the object is illuminated; the characteristics of the surface material determine the direction, intensity, and the frequency of the reflected light. Some of the characteristics that determine reflected light are inherent in the object's shape (as defined by its geometry); other characteristics of an object's surface are controlled by the GL lighting model.

To return to the example, the GL lighting model permits both ways of making an object appear to be blue: you can define a blue light and shine it on a white object, or you can define a white light and shine it on a blue object.

9.2 Material Reflectance

In GL lighting, the material characteristics of an object determine how it reflects light. Once light reaches an object, it is reflected in three ways:

- diffuse, which shows up as a matte or flat reflection
- specular, which shows up as highlights
- ambient, which simulates indirect light

9.2.1 Diffuse Reflectance

Diffuse reflectance gives the appearance of a matte or flat reflection from an object's surface. The direction of the light as it falls on the surface determines how bright the surface's diffuse reflection is. Diffuse reflection is brightest when the incident light strikes perpendicular to the surface.

For example, consider a distant light source shining directly on the north pole of a sphere. The diffuse reflectance of the sphere causes the sphere to appear brightest at the north pole. The brightness falls off as you look farther down the sides of the sphere. South of the equator, there is no diffuse reflectance at all.

Because diffuse light reflects equally in all directions, the position of the viewer has no effect on an object's diffuse reflection.

9.2.2 Specular Reflectance

Specular reflectance creates highlights and is dependent on the position of the viewpoint. For example, consider the glare in a rearview mirror from the headlights of a car behind you. If you shift your head a few inches to the right or left, you cannot see the glaring headlights in your mirror.

The intensity of specular reflection is typically highest along the direct angle of reflection.

9.2.3 Ambient Reflectance

Diffuse and specular reflectance simulates how objects in the scene reflect light that comes directly from a light source. Ambient reflectance, on the other hand, simulates light reflected from other objects in the scene, rather than directly from the light source. For example, if you look under your desk (presuming that you have a light on your desk that does not shine directly under it), you can still see things, even though the area under your desk is not directly illuminated. In reality, this ambient light is reflected from other surfaces in the room.

The ambient component is most noticeable on portions of the object that receive no direct illumination.

9.2.4 Emission

The final lighting component that objects can have is an emitted component. Emission is fairly limited in use. For one thing, an object that emits light does not also act as a light source: it does not add illumination to a scene. Furthermore, emission is featureless: a sphere that emits light appears as a featureless disc in the scene because it emits light equally in all directions. Adding an emitted component is useful for simulating the appearance of lights in the scene.

9.3 Setting Up GL Lighting

This section introduces fundamental GL lighting concepts such as surface normals, material properties, and light sources. First, it covers all that you need to create a static lighting environment. Section 9.4 discusses methods to change the lighting characteristics dynamically. Section 9.5 covers additional basic topics. Beyond these basics, section 9.6 describes advanced topics such as spotlights and high-performance lighting. Code fragments from real programs are used extensively to encourage a learn-by-example approach. A complete program is included at the end of the chapter.

9.3.1 Surface Normals

Surface normals are unit-length vectors that are perpendicular to a given surface. They are required for GL lighting. They serve as input to the lighting formula.

The GL maintains a current normal. The current normal is analogous to the current color. It stays the same until changed.

Here is an example of a point with a normal:

```
static float np[3] = {0, .7071, .7071};
static float vp[3] = {0, 0, -1};

    bgnpoint();
    n3f(np);
    v3f(vp);
    endpoint();
```

You can also provide normals at each vertex.

Here is an example of a triangle with normals:

```
static float np[3][3] =
    {{-.08716, 0, .9962}, {.08716, 0, .9962}, {0, 0, 1}};
static float vp[3][3] =
    {{-.08716, 0, .9962}, {.08716, 0, .9962}, {0, .1, 1}};

    bgnpolygon();
    n3f(np[0]);
    v3f(vp[0]);
    n3f(np[1]);
    v3f(vp[1]);
    n3f(np[2]);
    v3f(vp[2]);
    endpolygon();
```

The relationship between the order of the vertexes and the direction of the normal is significant. When the order of the vertexes of a triangle is viewed as counterclockwise, the triangle is facing the viewer. In this case, the normals of the triangle should also face the viewer. The triangle in the example follows this convention. This is often referred to as the *right-hand rule*. The right-hand rule makes it possible to distinguish between the front and the back sides of the triangle. Although it is not always necessary to follow this rule when using lighting, it allows you to use backface elimination. A feature called two-sided lighting, described later, requires you to follow this rule.

Points, lines, polygons, and character strings can all be lighted. As a general rule, any GL primitive that uses the current color can also be lighted. The lighting calculation is performed at each vertex of a primitive.

When using GL NURBS surfaces (see Chapter 14, "Curves and Surfaces"), normals can be generated automatically from the surface descriptions.

9.3.2 Setting Up Lighting Components

You configure three lighting components: the material, the light sources, and the lighting model. The material represents a set of properties that determines how it behaves under illumination. Each light source has a position and a light color. The lighting model defines the characteristics of the lighting environment. The combination of these three components determines the appearance, or more specifically the color, at each lighted vertex.

You configure each of these three components using a two-step process. The first step is one of *definition* using `lmdef`. The second is one of *activation* using `lmbind`. This two-step process is analogous to defining and setting a pattern with `defpattern` and `setpattern`.

The first example configures a material. First, we define a material in terms of a set of properties. Here is an example of a greenish plastic-like material:

```
static float mat[] = {
    AMBIENT, .1, .1, .1,
    DIFFUSE, 0, .369, .165,
    SPECULAR, .5, .5, .5,
    SHININESS, 30,
    LMNULL
};

    lmdef(DEFMATERIAL, 39, 0, mat);
```

Note that an array of type `float` holds a sequence of material properties. This sequence is terminated by the special constant `LMNULL`, which must always be the last float in a property array. Each property expects a fixed number of floats to follow it. For example, the `DIFFUSE` property is followed by three floats that are the red, green, and blue diffuse reflectance coefficients. Similarly, the `AMBIENT` and `SPECULAR` properties are each followed by three floats which represent the ambient and specular reflectance coefficients, respectively. All color components should be in the range 0 to 1.

`SHININESS` is followed by one float that controls the size and apparent brightness of a specular highlight. The shininess value can range from 0 to 128. Higher values result in smaller, more focused specular highlights. A shininess value of 0 is special, and disables specular reflection entirely.

`lmdef` associates the properties in a property array with an integer index and copies these properties at the time of the call. The first argument to `lmdef` determines whether the properties in the array apply to a material, a light source, or a lighting model. In this example, the `lmdef` call remembers the material properties in the `mat` array. This creates a material definition that will be referenced later by its integer index, 39, specified by the second parameter to `lmdef`. The third parameter has an obscure purpose and is ignored when set to zero. For simplicity, the examples in this section always set it to zero.

The next example defines a slightly reddish-colored light source:

```
static float lt[] = {
    LCOLOR, 1, .8, .8,
    POSITION, 0, 1.5, -.5, 1,
    LMNULL
};

    lmdef(DEFLIGHT, 27, 0, lt);
```

Although you use the same syntax for a light source as for a material, the actual properties are different. In this case, the `LCOLOR` property is followed by three floats that are the red, green, and blue components of the light source color. A light source is normally omnidirectional — that is, it emits light of equal intensity in all directions.

`POSITION` is followed by four floats that are the *x*, *y*, *z*, and *w* coordinates of the light source. A light source position is defined in *homogeneous coordinates*. Whenever the *w* component of the position is non-zero, we are defining a point light source. For our purposes, when the *w* coordinate is one, the *x*, *y*, and*z* coordinates simply represent the three-dimensional position of the light source. The current matrix does not affect the light source position at the time of this `lmdef` call. We will discuss later how the `POSITION` property is transformed by current matrix. This light source definition will be referenced later by its integer index, 27.

The next example defines a typical lighting model:

```
static float lm[] = {
    AMBIENT, .1, .1, .1,
    LOCALVIEWER, 1,
    LMNULL
};
```

```
lmdef(DEFLMODEL, 14, 0, lm);
```

Once again, you use the same syntax for a lighting model as for a material or light source. The AMBIENT property specifies the color of ambient light. This ambient light is non-directional. It interacts only with the ambient reflectance of a material.

Specular reflection from a point on a surface depends on the normal, the direction to the light source, and the direction to the viewpoint. The LOCALVIEWER property is a flag that determines whether the direction to the viewpoint needs to be recalculated at each vertex. It is followed by a single float. A value of 1 indicates that the flag is on and a value of 0 indicates that the flag is off. Turning on the LOCALVIEWER flag places the viewpoint at the origin looking down the negative z axis. More about LOCALVIEWER is described in section 9.5.2.

At this point, we have defined the three components of lighting. This completes the first step. The second step is to activate these components. Before activating any lighting definitions, you must be in multi-matrix mode:

```
mmode(MVIEWING);
```

The next example shows how to activate, or *bind*, a material, a light source, and a lighting model.

```
lmbind(MATERIAL, 39);
lmbind(LIGHT0, 27);
lmbind(LMODEL, 14);
```

Material index 39 represents the material definition from the example above. You activate this material definition by binding it to the target MATERIAL. Only one material can be active at any time. Light source 27 represents the light source definition from a previous example. It is made active by binding it to the target LIGHT0.

The current ModelView matrix transforms the position of the light source when you call `lmbind` (see Chapter 7, "Coordinate Transformations"). In this example, the position of the light source has a non-zero *w* component. This makes it a point light source. Its position is transformed in the same way that a point is transformed.

There are at least eight light source targets available, named LIGHT0, LIGHT1, LIGHT2, and so on. The actual number of these targets is defined in *gl.h* by the constant MAXLIGHTS. Any of the light source targets can be active at any time. You can bind a light definition to only one light source target.

Lighting model index 14 represents the lighting model definition from a previous example. You activate it by binding it to the target LMODEL. Only one lighting model can be active at any time.

At this point, lighting is enabled. More specifically, lighting is enabled when both a material and a lighting model are bound. The indexes 39, 27, and 14 were chosen for this example and are arbitrary. You could use index 1 for all three components because indexes for materials are separate from those of lights, and so on. Any index between 1 and 65535 is legal. Index 0 has a special purpose and will be described in the next section.

9.4 Changing Lighting Settings

Now that we have discussed how to turn lighting on, it is only natural to ask how to turn it off. As mentioned previously, lighting is enabled when both a material and a lighting model are bound. To turn off lighting, simply unbind either the material or the lighting model. Here is one way to turn lighting off:

```
lmbind(LMODEL, 0);
```

Index 0 has a special purpose. You can only use it with lmbind. Index 0 is used to unbind a material, light, or lighting model.

This example turns off a light source:

```
lmbind(LIGHT0, 0);
```

Assume that lighting is on at this point. Note that after this lmbind call, lighting remains on because the presence of active lights is not necessary for lighting.

Suppose you want to use more than one material for some geometry that you are drawing. Assume that you have defined a second material . When you bind this second material definition, it becomes the active material, overriding the previously bound material.

Suppose you want to change an existing definition instead of creating a new definition for each variation. You can use `lmdef` to change the properties of an existing definition.

Recall the point light source definition 27 from before. Here is an example of changing this light source definition to produce a greenish color.

```
static float lt[] = {
    LCOLOR, .8, 1, .8,
    LMNULL
};

    lmdef(DEFLIGHT, 27, 0, lt);
```

If light source definition 27 is currently bound to `LIGHT0`, then the light color change takes effect immediately. Because `LCOLOR` is the only property in this array, only this property is changed.

A light source position is transformed by the ModelView matrix at the time of the `lmbind` call. This transformed position remains. It can be retransformed by binding it again. An active light source can only be boundto the same light target. This is often used to make a light move from frame to frame.

9.5 More Lighting Features

Sections 9.3 and 9.4 discussed the components for basic lighting. This section covers additional topics that are still at a basic level.

9.5.1 Infinite Lights

Recall the POSITION property that is used to define a light source. A light source position is defined in homogeneous coordinates. If the *w* coordinate is zero, then the *x*, *y*, and*z* coordinates represent a light source direction. This is called an *infinite light source* and can be a good approximation for a distant light source.

Here is an example of a white light source that is positioned infinitely far away on the positive *z* axis:

```
static float lt[] = {
    LCOLOR, 1, 1, 1,
    POSITION, 0, 0, 1, 0,
    LMNULL
};

    lmdef(DEFLIGHT, 27, 0, lt);
```

Note that the light source direction points from the vertex to the light source. The GL normalizes the light source direction.

Infinite light sources have a performance advantage over point light sources.

9.5.2 Infinite Viewpoint

Earlier, we showed how to place the viewpoint at the origin. This is called a *local viewpoint.* You can also place the viewpoint infinitely far away on the positive *z* axis.

Here is a lighting model with an infinite viewpoint:

```
static float lm[] = { LOCALVIEWER, 0, LMNULL };

    lmdef(DEFLIGHT, 14, 0, lm);
```

Setting an infinite viewpoint has a performance advantage. In some cases, the results look identical to a local viewpoint because the viewpoint is needed only to determine the specular reflection. Actually, if the SPECULAR material property is set to all zeroes, a local viewpoint has no benefit. By default, the viewpoint is infinite.

When you use an infinite viewpoint with infinite lights, primitives with the same normal and material properties produce identical colors. In practice, this means that flat surfaces are lighted with constant color, which might appear slightly unrealistic.

9.5.3 Ambient Light and Emission

Section 9.3.2 shows how you can specify an ambient light source as part of the lighting model. You can also specify additional ambient light with each light source. The syntax for this is the same as for the lighting model.

The ambient light source color is multiplied by the ambient material property. This product is one component of the color computed by lighting.

The emission material property is another component of this computed color.

Here is an example of a material that only has emission:

```
static float mat[] = {
    EMISSION, 0, .369, .165,
    LMNULL
};

    lmdef(DEFMATERIAL, 39, 0, mat);
```

9.5.4 Non-Unit-Length Normals

Earlier we mentioned that normals must be unit-length. This is true by default. The GL does have the ability to handle non-unit-length normals. There is often a performance penalty for doing this.

Here is how to configure the GL to normalize normals automatically:

```
    nmode(NNORMALIZE);
```

Lighting does not have to be on to set this mode. This feature can be disabled with:

```
nmode(NAUTO);
```

This is the default.

9.6 Advanced Lighting Features

This section covers advanced lighting features. Not all lighting features are supported on all systems. Consult the *Graphics Library Reference Manual* to determine whether a feature is available.

9.6.1 Attenuation

In reality, the effect of a light source on a surface diminishes as the distance between the light source and the surface increases. You can simulate this effect with the attenuation feature of GL lighting. Attenuation is defined inthe lighting model and applies to all point light sources. Attenuation does not apply to infinite or ambient light sources.

Attenuation is a function of the distance between a point light source and the surface it illuminates. The formula for attenuation is:

$$attenuation\,factor = 1 / (k0 + k1*dist + k2*dist*dist)$$

dist is the distance between the vertex and the point light source. This distance is never negative. *k0* controls constant attenuation, *k1* controls linear attenuation, and *k2* controls distance-squared attenuation. The attenuation formula is calculated for each lighted vertex.

Here is an example of adding linear attenuation to a previously defined lighting model with index 14:

```
static float atten[] = {
    ATTENUATION, .1, 1,
    LMNULL
};

    lmdef(DEFLMODEL, 14, 0, atten);
```

The ATTENUATION property is followed by two non-negative floats. These floats specify *k0* and *k1* of the attenuation formula.

Here is an example that adds distance-squared attenuation to lighting model 14:

```
static float atten[] = {
    ATTENUATION, .1, 0,
    ATTENUATION2, 1,
    LMNULL
};

    lmdef(DEFLMODEL, 14, 0, atten);
```

The ATTENUATION2 property is followed by one non-negative float.

The defaults are that $k0 = 1$, $k1 = 0$ and $k2 = 0$. These values define a lighting model without attenuation. You can disable attenuation by restoring these default values.

Both of these examples use a small but non-zero value for the constant term *k0*. As *dist* approaches zero, a non-zero *k0* bounds the maximum value of the attenuation formula, otherwise it would approach infinity.

9.6.2 Spotlights

It was mentioned earlier that a point light source is omnidirectional. This is true by default. You can make a point light source into a spotlight using the SPOTLIGHT property.

A spotlight emits a cone of light that is centered along the spotlight direction. The intensity of a spotlight is a function of the angle between the spotlight direction and the direction to the vertex being illuminated. Typically, the intensity falls off as this direction angle increases.

You can control the shape of a spotlight's intensity falloff with two values: an *exponent* and a *spread angle*. An exponent of 1 produces a gradual falloff that is actually the cosine of the direction angle. An exponent of 128 gives the sharpest possible falloff. An exponent of 0 gives a constant intensity. The spread angle defines a cone outside of which no light is emitted. The intensity falloff as controlled by the exponent is cut off by this cone. The cone defined by the spread angle is independent of the intensity falloff controlled by the exponent.

Here is an example that defines and binds a white spotlight:

```
static float spot[] = {
    LCOLOR, 1, 1, 1,
    POSITION, 0, 2, 0, 1,
    SPOTDIRECTION, 0, -1, 0,
    SPOTLIGHT, 100, 45,
    LMNULL
};

    lmdef(DEFLIGHT, 11, 0, spot);
    lmbind(LIGHT0, 11);
```

The SPOTLIGHT property is followed by two floats specifying the exponent and spread angle of the light cone. An exponent of 100 specifies a very sharp falloff. The exponent can range from 0 to 128. A spread angle of 45 defines a cone with a radius angle of 45 degrees. The spread angle can range from 0 to 90 degrees. A special value of 180 degrees is also permitted, and is used in the next example.

The SPOTDIRECTION property is followed by three floats, the *x*, *y*, and *z* coordinates of the spotlight direction vector. It is automatically normalized. This example points the spotlight in the direction of the negative *y* axis. The spotlight direction vector is transformed by the current ModelView matrix in the same manner as a normal.

Notice that the POSITION property specifies a point light source. This is necessary for a spotlight. The SPOTLIGHT property is ignored for an infinite light source.

Here is an example that turns off the spotlight effect:

```
static float spotoff[] = { SPOTLIGHT, 0.0, 180.0, LMNULL };

    lmdef(DEFLIGHT, 11, 0, spotoff);
```

The combination of setting the exponent to 0.0 and the spread angle to 180.0 turns off the spotlight effect. By default, a point light source is not a spotlight. The SPOTDIRECTION property is ignored while the light source is not a spotlight.

You can combine spotlights with attenuation to yield an effect that is reminiscent of a real spotlight. The spotlight effect can create a highly nonlinear intensity gradient across a surface. Ensure that the vertexes of a surface illuminated by a spotlight are closely spaced so that the approximation of this gradient is accurate.

9.6.3 Two-sided Lighting

In general, lighting calculations are correct only when you view the side of a polygon where the normal faces toward you. If you use the right-hand rule to define the polygons and their normals, lighting calculations are then correct for the front faces of those polygons. This is called one-sided lighting. With two-sided lighting, the backfacing polygons are also correct.

Here is an example that adds two-sided lighting to a lighting model definition:

```
static float two[] = { TWOSIDE, 1, LMNULL };

    lmdef(DEFLMODEL, 14, 0, two);
```

The TWOSIDE property is followed by one float, either 1 or 0, that specifies whether the two-sided lighting feature should be enabled or disabled, respectively. It is disabled by default.

Two-sided lighting applies only to primitives with facets, such as polygons or triangle meshes. Two-sided lighting is ignored for other lighted primitives, such as points or lines.

With two-sided lighting, the material properties of the front and back faces are normally identical — that is, the active material is used for both the front and the back. You can also specify independent front and back material properties. Independent front and back materials can be useful to distinguish between the inside and the outside of an object. This feature is quite effective when combined with user-defined clipping planes. See Chapter 8, "Hidden Surface Removal."

This example binds material definition 40 to the back material as follows:

```
lmbind(BACKMATERIAL, 40);
```

You unbind the back material by binding it to 0:

```
lmbind(BACKMATERIAL, 0);
```

By default, the back material is bound to 0. Whether a back material is bound or not has no effect on whether lighting is on or off.

9.6.4 Fast Updates to Material Properties

We have already learned how to change the properties of an existing definition using `lmdef`. When you change an active definition, those changes take effect immediately.

Assume that material definition 39 exists. The following example changes its `DIFFUSE` property:

```
static float mat[] = {
    DIFFUSE, .369, 0, .165,
    LMNULL
};

    lmdef(DEFMATERIAL, 39, 0, mat);
```

Once again, if this material is currently bound, the change takes effect immediately. This mechanism provides a general method for changing the properties of a material, a light source, or a lighting model.

An efficient method of changing material properties can be desirable. For example, it is reasonable to change a specific material property at each vertex of many polygons. For this reason, the GL provides a higher performance method of updating material properties.

`lmcolor` sets a mode where the current color updates a specific material property. The current color should be set explicitly after the call to `lmcolor` and before a vertex or normal is issued. The current color can be set with `c`, `cpack`, or `RGBcolor`.

Some of the color commands specify red, green, blue, and alpha as integers in the range of 0 to 255. This range is mapped to a 0.0 to 1.0 range when used to update material properties.

Here is an another method of updating the `DIFFUSE` property of the current material.

```
lmcolor(LMC_DIFFUSE);
RGBcolor(94, 0, 42);
lmcolor(LMC_COLOR);
```

The first call to `lmcolor` sets `LMC_DIFFUSE` mode. In this mode, the `RGBcolor` call directly updates the diffuse property. The second call to `lmcolor` restores the default mode.

This example sets the diffuse property of the active material to roughly the same values as the previous example, but there is an important difference. When you use `lmdef` to change an active material, the material definition also changes. When you use `lmcolor` to change an active material, the change has no effect on the material definition. Actually, any changes made to the active material using `lmcolor` are lost when you use `lmbind` to bind another material.

Here is an example of updating the ambient and diffuse properties of the current material at each vertex of a polygon.

```
lmcolor(LMC_AD);
bgnpolygon();
cpack(0x800000ff);
n3f(np[0]);
v3f(vp[0]);
cpack(0x8000ff00);
n3f(np[1]);
v3f(vp[1]);
cpack(0x80ff0000);
n3f(np[2]);
v3f(vp[2]);
endpolygon();
lmcolor(LMC_COLOR);
```

This example uses a normal array, `np`, and a vertex array, `vp`, in the same manner as the triangle example in section 9.3.1. The call to `lmcolor(LMC_AD)` sets a mode where the calls to `cpack` directly update the ambient and diffuse material properties simultaneously.

The two modes `LMC_AD`, which updates both the ambient and diffuse properties, and `LMC_DIFFUSE`, which updates only the diffuse property, also update the `ALPHA` material property with the alpha component of the current color. Note that the alpha component in this example is set to `0x80` (roughly 0.5) at each vertex.

The default mode, LMC_COLOR, has an interesting property. If no normals are present for a primitive, that primitive is not lighted. More exactly, if a color command follows the last normal before a primitive is drawn, that primitive is not lighted.

LMC_EMISSION, LMC_AMBIENT, and LMC_SPECULAR update their corresponding material properties.

In LMC_NULL mode, color commands are ignored while lighting is enabled.

If two-sided lighting is enabled and no BACKMATERIAL is bound, then such changes to material properties will affect both the front and back faces. If a BACKMATERIAL is bound, changes to material properties affect only the front face.

9.6.5 Default Settings

The following example sets the defaults for lighting:

```
lmdef(DEFMATERIAL, 39, 0, NULL);
lmdef(DEFLIGHT, 27, 0, NULL);
lmdef(DEFLMODEL, 14, 0, NULL);
```

Passing NULL as the fourth parameter creates a definition of a default set of properties. Also, it is equivalent to pass an array of type `float` that has one element, the special constant LMNULL.

When you create a definition with lmdef, the properties are first set to their default values. Properties specified in the array override these defaults. Later, when you change this definition with `lmdef`, properties not specified in your array are left unchanged.

These definitions contain the default settings for material, light source, and lighting model:

```
static float mat[] = {
    EMISSION, 0, 0, 0,
    AMBIENT, .2, .2, .2,
    DIFFUSE, .8, .8, .8,
    SPECULAR, 0, 0, 0,
    SHININESS, 0,
    ALPHA, 1,
```

```
    COLORINDEXES, 0, 127.5, 255,
    LMNULL
};

static float lt[] = {
    AMBIENT, 0, 0, 0,
    LCOLOR, 1, 1, 1,
    POSITION, 0, 0, 1, 0,
    SPOTLIGHT, 0, 180,
    SPOTDIRECTION, 0, 0, -1,
    LMNULL
};

static float lm[] = {
    AMBIENT, .2, .2, .2,
    LOCALVIEWER, 0,
    TWOSIDE, 0,
    ATTENUATION, 1, 0,
    ATTENUATION2, 0,
    LMNULL
};
```

9.6.6 Transparency

Normally, materials are opaque. You can control the transparency of a material with alpha. In lighting, alpha is specified in the material definition.

```
static float mat[] = {
    ALPHA, .5,
    LMNULL
};
```

The ALPHA property is followed by one float. When used in conjunction with `blendfunction`, a transparent effect can be achieved. The use of LMC_DIFFUSE or LMC_AD mode overrides the ALPHA material property with the alpha of the current color.

The ALPHA property works with two-sided lighting. You can achieve different front and back transparencies by binding material definitions with different ALPHA properties to MATERIAL and BACKMATERIAL.

9.6.7 Lighting With Multiple Windows

You can use lighting in more than one GL window. The definitions that `lmdef` creates and modifies are shared among all windows. On the other hand, the material, light sources, and lighting model that `lmbind` activates are specific to the window that was active at the time of the `lmbind` call.

9.6.8 Restrictions on ModelView and Projection Matrices

Many GL programs change the ModelView matrix using only `rot`, `rotate`, `scale`, and `translate`. They change the projection matrix using only `ortho2`, `ortho`, `perspective`, and `window`. These calls all work with lighting. `loadmatrix` and `multmatrix` let you specify a general 4 x 4 matrix. Certain restrictions apply to this general matrix when lighting is used.

No projection components are allowed in the ModelView matrix. More specifically, the rightmost column of the matrix must be `[0 0 0 1]`.

Two restrictions apply to the projection matrix. First, no rotation is allowed. Second, the top two elements of the right column must be 0. The IRIS-4D VGX permits a general 4 x 4 projection matrix, and so these restrictions do not apply.

9.7 Lighting Performance

This section gives a general feeling for the performance implications of specific lighting features. The performance of a given feature might vary between the different graphics products as well as between different software releases on the same product. Nevertheless, there are some guidelines you can follow.

Each additional light source takes extra computation. The more you use, the longer it takes to compute the color for a vertex.

A difficult calculation that can occur in lighting is the square root operation. It is used for a local viewpoint, a point light source, and in normalizing non-unit-length normals. The use of any of these features adversely affects performance.

In some cases, normals transformed by the ModelView matrix will not maintain their unit length. The GL detects this condition of the matrix and automatically renormalizes the transformed normals. To avoid this extra calculation, use `scale(x, y, z)` only if $x = y = z$. Similarly, ensure that you use an orthonormal matrix with `loadmatrix` or `multmatrix`.

The two-sided lighting feature takes extra computation, but its performance should be better than half the performance of one-sided lighting.

The use of GL calls other than `n`, `v`, `c`, `cpack`, and `RGBcolor` between `bgn` and `end` calls might incur a performance penalty. In fact, only a limited set of calls are allowed between a `bgn` and `end` call. This list is included in the *man* page for `bgnpolygon`.

The use of `lmcolor` with an argument other than `LMC_COLOR` or `LMC_NULL` might incur a slight performance penalty.

A natural question to ask is, "how do I get the highest possible performance from lighting?" Here are some suggestions:

- use an infinite viewpoint by setting `LOCALVIEWER` to 0, the default
- use a single infinite light source
- use the default `lmcolor(LMC_COLOR)` or use `lmcolor(LMC_NULL)`
- use the default `nmode(NAUTO)`

- take advantage of drawing primitives that share vertices for lines or polygons; for instance, use `bgntmesh` or `bgnqstrip` for polygon drawing

On IRIS-4D VGX Series systems, the use of less than one normal per vertex (such as one normal per polygon) does not reduce the lighting calculation specifically. However, it does reduce the amount of data transferred to the geometry engines.

9.8 Color Map Lighting

There is a way to do lighting in color map mode, but it is designed for systems without enough bit planes to support RGB mode. On graphics systems with enough bitplanes, RGB mode lighting is recommended.

Color map lighting generates a pseudo-intensity which is a function of the direction to the light source and the direction to the viewpoint. This pseudo-intensity is mapped to a color map value. A well-chosen range of color map values gives a reasonable lighting effect. You can represent multiple materials by creating a color map range for each material. Color map lighting is enabled when both lighting and color map mode are enabled.

Color map lighting has some inherent limitations, and many of the advanced lighting features are not supported. Color map lighting recognizes only a limited set of properties, described here.

A material definition uses only the `COLORINDEXES` property.

```
static float mat[] = {
    COLORINDEXES, 512, 576, 639,
    LMNULL
};
```

This property is followed by three floats, representing an ambient index, a diffuse index, and a specular index. These indices should correspond to appropriate values in the color map. Lighting produces values that range from the ambient index to the specular index. Lighting generates the ambient index when there is no diffuse or specular reflection. It generates the diffuse index when the diffuse reflection is at a maximum, but there is no specular reflection. It generates the specular index when the specular reflection is at a maximum. The specular index must be greater than or equal to the diffuse index, which must in turn be greater than or equal to the ambient index. All other material properties are ignored.

A light source definition uses only the `POSITION` property.

```
static float lt[] = {
    POSITION, 0, 0, 1, 0,
    LMNULL
};
```

Only infinite light source positions are allowed. This requires that you set the *w* component of `POSITION` to 0. All other light source properties are ignored. You can specify zero or more light sources. Each light source contributes its full intensity.

A lighting model definition uses only the `LOCALVIEWER` property.

```
static float lm[] = {
    LOCALVIEWER, 0,
    LMNULL
};
```

Only an infinite viewpoint is allowed. `LOCALVIEWER` can only be set to 0, which is the default.

9.9 Sample Lighting Program

The following example program demonstrates the GL lighting facility.

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

Matrix Identity =
    { 1, 0, 0, 0,  0, 1, 0, 0,  0, 0, 1, 0,  0, 0, 0, 1 };

float mat[] = {
    AMBIENT, .1, .1, .1,
    DIFFUSE, 0, .369, .165,
    SPECULAR, .5, .5, .5,
    SHININESS, 10,
    LMNULL,
};

static float lm[] = {
    AMBIENT, .1, .1, .1,
    LOCALVIEWER, 1,
    LMNULL
};

static float lt[] = {
    LCOLOR, 1, 1, 1,
    POSITION, 0, 0, 0, 1,
    LMNULL
};

main()
{
long xorig, yorig, xsize, ysize;
float rx, ry;
short val;
```

```
    winopen("cylinder");
    getorigin(&xorig, &yorig);
    getsize(&xsize, &ysize);
    RGBmode();
    doublebuffer();
    gconfig();
    lsetdepth(0, 0x7fffff);
    zbuffer(1);
    mmode(MVIEWING);
    loadmatrix(Identity);
    perspective(600, xsize/(float)ysize, .25, 15.0);
    lmdef(DEFMATERIAL, 1, 0, mat);
    lmdef(DEFLIGHT, 1, 0, lt);
    lmdef(DEFLMODEL, 1, 0, lm);
    lmbind(MATERIAL, 1);
    lmbind(LMODEL, 1);
    lmbind(LIGHT0, 1);
    translate(0, 0, -4);

    while (!getbutton(ESCKEY)) {
        ry=300*(2.0*(getvaluator(MOUSEX)-xorig)/xsize-1.0);
        rx=-300*(2.0*(getvaluator(MOUSEY)-yorig)/ysize-1.0);
        czclear(0x404040, 0x7fffff);
        pushmatrix();
        rot(ry, 'y');
        rot(rx, 'x');
        drawcyl();
        popmatrix();
        swapbuffers();
    }
}

drawcyl()
{
double dy = .2;
double theta, dtheta = 2*M_PI/20;
double x, y, z;
float n[3], v[3];
int i, j;
```

```
    for (i = 0, y = -1;  i < 10;  i++, y += dy)  {
        bgntmesh();
        for (j=0, theta=0;  j<=20;  j++, theta+=dtheta)  {
            if (j == 20)  theta = 0;
            x = cos(theta);
            z = sin(theta);
            n[0] = x;  n[1] = 0;  n[2] = z;
            n3f(n);
            v[0] = x;  v[1] = y;  v[2] = z;
            v3f(v);
            v[1] = y + dy;
            v3f(v);
        }
        endtmesh();
    }
}
```

10. Pixels

Pixels, like raster fonts, are not nearly as easy to transform (rotate, scale, etc.) as geometric figures. Code that reads and writes pixels on the screen often knows (or has to find out) something about the window dimensions, the screen resolution, and so on.

Another problem with reading and writing pixels is that the contents of each pixel can mean different things depending on the display mode for that pixel. The same physical bitplanes are used to store either color map indices or RGB values; accordingly, the mode of the pixel determines whether the contents are interpreted as RGB triples or as indices into the color map.

The IRIS-4D supports three styles of pixel access:

The first provides a high-performance interface to a flexible set of pixel routines—`lrectwrite`, `lrectread`, `rectcopy` `pixmode`, and `rectzoom` These routines operate on arbitrarily sized rectangles made up of arbitrarily sized pixels. The behavior of these routines can be modified by setting parameters for offset, stride, pixel size, zoom, and other features described later. Note that these functions operate in either RGB or color map mode.

The second style is compatible with older versions of the Graphics Library. It provides a high-performance interface to operate on arbitrarily sized rectangles. This set of routines includes `rectwrite`, `rectread`, and `rectzoom`. The behavior of these routines can be modified only for zoom. Note that these functions operate in either RGB or color map mode, but because they operate with 16-bit pixels, they are not generally useful for RGB mode.

The third is compatible with even older versions of the Graphics Library. It provides mode-dependent routines to deal with, at most, one scanline at a time, which is positioned according to the system's "current character position". They include `writepixels`, `writeRGB`, `readpixels`, and `readRGB`. Continued use of these routines is not suggested because higher-performance, more flexible routines are available.

10.1 Pixel Formats

The following formats constitute the standard Silicon Graphics pixel formats:

- RGBA (Red-Green-Blue-Alpha) data is interpreted as four 8-bit values packed into each 32-bit word. Bits 0-7 represent red, bits 8-15 represent green, bits 16-23 represent blue, and bits 24-31 represent alpha.

  For example, 0X01020304 corresponds to a pixel whose RGBA values are 4, 3, 2, and 1, respectively. This is exactly the same format `cpack` uses. (See Chapter 4, "Display and Color Modes," for more information.)

- CI (Color Index) data is interpreted as 12-bit (low order) indices into a single, 4096-entry color map. The high-order 20 bits should be zero.

- *z*-buffer data is interpreted as 24-bit (low order) data. The high-order 8 bits should be zero.

- Other data used in overlay, underlay, and pop-up planes is interpreted as a type of color map data. Because different systems and different configurations support different pixel sizes for these resources, the number of entries contained in the auxiliary color map vary.

All systems in all configurations support the pixel formats listed above.

The pixel formats described above are frame buffer formats. The format of the pixel data in host memory can be packed in a more efficient format if `pixmode` is used with `lrectread` and `lrectwrite`. Default behavior is backward-compatible and therefore unpacked. `pixmode` features are described later in this chapter.

10.2 Pixel Sources and Destinations

The following routines establish sources and destinations for pixel operations, as well as other drawing routines. You can specify more than one destination. Sources apply to reads and copies; destinations apply to writes and copies.

readsource

`readsource`(*rsource*) determines the source of pixels read by `rectread`, `lrectread`, `rectcopy`, `readpixels`, and `readRGB`. The default value is SRC_AUTO, which selects the front buffer in single buffer mode and the back buffer in double buffer mode. SRC_FRONT always reads from the front buffer (this is always valid), and SRC_BACK always reads from the back buffer (valid only in double buffer mode). SRC_ZBUFFER reads 24-bit data from the *z*-buffer. Other sources such as SRC_FRAMEGRABBER are available if special hardware is installed.

drawmode, frontbuffer, backbuffer, zdraw()

`drawmode`(*mode*) determines the drawing mode, hence, the destination for pixel operations. NORMALDRAW is the default; OVERDRAW, UNDERDRAW and PUPDRAW are options. In NORMALDRAW mode, `frontbuffer`(*frontflag*), `backbuffer`(*backflag*), and `zbuffer`(*zflag*) apply. If you assert more than one destination in NORMALDRAW mode, more than one destination is written.

10.3 Reading/Writing Pixels Efficiently

This section describes subroutines that read and write pixels with the highest possible performance. No color mode checking is done, so RGB data written in color map mode, as well as the reverse, can return undesired results.

rectread and lrectread

`rectread`(*x1*, *y1*, *x2*, *y2*, *sarray*) reads a rectangular array of pixels from the window where (*x1*,*y1*) are the coordinates for the lower-left corner of the rectangle and (*x2*,*y2*) are the coordinates for the upper-right corner. All coordinates are relative to the lower-left corner of the window in screen coordinates. *sarray* is an array of 16-bit values. Only the low-order 16 bits of each pixel are read, so `rectread` is useful primarily for windows drawn in color map mode. The data is loaded into *sarray* left to right and bottom to top. In other words, if the pixel data on the screen looks like this:

```
1  2  3  4
5  6  7  8
9 10 11 12
```

sarray contains {9,10,11,12,5,6,7,8,1,2,3,4}, i.e., *sarray*[0]=9, *sarray*[1]=10, etc. `rectread` returns the number of pixels successfully read. Normally, this is:

```
(x2 - x1 + 1) * (y2 - y1 + 1)
```

If any part of the specified rectangle is off the screen, or if the coordinates are mixed up, the behavior of `rectread` is undefined.

Errors occur only outside the screen, not outside the window. It is possible to read pixels outside a window, as long as they are on the physical screen. This can be useful for certain applications that magnify data from other windows, or do image processing on images produced by other programs. The main difficulty is that the data can come from areas of the screen that are in different color modes (color map or RGB mode). Because `rectread` is not restricted to the current window, any or all of the coordinates can be negative.

`lrectread`(*x1*, *y1*, *x2*, *y2*, *larray*) is similar to `rectread` except that *larray* contains 32-bit quantities and the behavior of `lrectread` is affected by `pixmode` settings. Using `pixmode` with `lrectread` provides useful data manipulation functions as well as data packing functions to store and transfer data more efficiently.

rectwrite and lrectwrite

`rectwrite`(*x1*, *y1*, *x2*, *y2*, *sarray*) writes a rectangular array of pixels to the window, where (*x1*,*y1*) are the coordinates for the lower-left corner of the rectangle and (*x2*,*y2*) are the coordinates for the upper-right corner. All coordinates are relative to the lower-left corner of the window in screen coordinates. *sarray* is an array of 16-bit values. Only the low-order 16 bits of each pixel are written, so `rectwrite` is useful primarily for windows in color map mode. The data is written from *sarray* from left to right and bottom to top as described above. `rectwrite` obeys the zoom factors set by `rectzoom` (see `rectzoom` below). `writemask` and `scrmask` apply as with other drawing primitives.

`lrectwrite`(*x1*, *y1*, *x2*, *y2*, *larray*) is similar to `rectwrite` except that *larray* contains 32-bit quantities and the behavior of `lrectwrite` is affected by `pixmode` settings. Using `pixmode` with `lrectwrite`() provides useful data manipulation functions as well as data unpacking functions to store and transfer data more efficiently.

rectcopy

`rectcopy`(*x1*, *y1*, *x2*, *y2*, *newx*, *newy*) copies the pixels from a rectangular region of the screen to a new region. As with `rectread` and `lrectread`, the source rectangle must be on the physical screen, but not necessarily constrained to the current window. The bitplane source is determined by `readsource` and the bitplane destination is determined by `drawmode`, `frontbuffer`, `backbuffer`, and `zdraw`.

During a `rectcopy`, the source rectangle can be zoomed by parameters established with `rectzoom`. In addition, data manipulation and mirroring can be accomplished through `pixmode` settings.

rectzoom

`rectzoom`(*xzoom*, *yzoom*) sets the independent *x* and *y* zoom factors for `rectwrite`, `lrectwrite`, and `rectcopy`. *xzoom* and *yzoom* are positive floating point values. Values less than 1.0 are allowed and cause rectangles to shrink. Some hardware platforms are not capable of providing non-integer zoom, so only the integer portion of the zoom parameters apply in that case.

If the following rectangle is copied after calling `rectzoom`(2.0, 3.0):

```
1 2
3 4
```

the following copy is made:

```
1 1 2 2
1 1 2 2
1 1 2 2
3 3 4 4
3 3 4 4
3 3 4 4
```

The following program is a simple magnification program. It magnifies the rectangular area above and to the right of the cursor to fill the window.

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

main()
{
    long xsize, ysize, readxsize, readysize, x, y;
    long xorg, yorg;

    winopen("zoom");
    getsize(&xsize, &ysize);
    getorigin(&xorg, &yorg);
    readxsize = xsize/3;
    readysize = ysize/3;
    rectzoom(3.0, 3.0);
    while (1) {
        x = getvaluator(MOUSEX);
        y = getvaluator(MOUSEY);
        rectcopy(x-xorg,y-yorg,x-xorg+readxsize,
            y-yorg+readysize, 0, 0);
    }
}
```

After determining the size and shape of the window, the program simply loops, copying an appropriately sized rectangle above and to the right of the cursor into the window magnified by a factor of 3 in each direction. The expressions *x-xorg* and *y-yorg* convert the cursor's screen coordinates into window coordinates.

To be a useful tool, the program should have a mechanism to change to and from RGB mode, perhaps a method to change zoom factor, and perhaps some code to avoid `rectcopy` if the mouse has not moved since the last time. It might also make a better user interface if the region around the cursor is magnified rather than the area above and to the right of it. Using this tool as is, note that regions of the screen drawn in RGB mode appear incorrect, and color-mapped portions look fine. Also, notice that with double-buffered programs, the zoom window appears to blink. This is caused by buffer swapping in the double-buffered program, while zoom is always reading from the same buffer. If the zoom window is magnified, a zoom recursion takes place and the effects are interesting.

10.4 Using pixmode

`pixmode` allows you to customize pixel operations when you use `lrectwrite`, `lrectread`, or `rectcopy`. It can be used to specify pixel operations such as shifting, expansion, offsetting, and packing and unpacking. Each function is selected by calling `pixmode` with an argument indicating the operation and a value for that operation. Any or all functions can be used in combination. Once you select a `pixmode` operation, it remains in effect until you change it. `pixmode` operations have no effect on `rectread` or `rectwrite` or any of the old-style pixel access subroutines.

10.4.1 Shifting Pixels

32-bit pixel data can be shifted left or right before being written to a frame buffer destination or before being read into memory using the `pixmode` operation PM_SHIFT. For example, to shift all pixels written with `lrectwrite` left by eight bits so that the red byte is placed in the green byte's position, the green byte is placed in the blue byte's position, the blue byte is placed in the alpha byte's position, and the alpha byte is lost, call:

```
pixmode(PM_SHIFT, 8);
```

before calling `lrectwrite` to write the pixel data. To disable shifting after you have enabled it, use:

```
pixmode(PM_SHIFT, 0);
```

You can achieve the same effect when copying pixel data using `rectcopy`. The same call also affects `lrectread`, but in this case a right shift is indicated with a negative value. Thus, if you perform the sequence of operations:

```
pixmode(PM_SHIFT, 8);
lrectwrite();
```

then call `lrectread` to read the pixels you just wrote, the pixels you get are exactly the same (unshifted) pixels you wrote, but with the alpha byte stripped off. This is because the pixels were shifted left eight bits when they were written and then shifted right eight bits when they were read back.

The allowable values for PM_SHIFT are 0, ±1, ±4, ±8, ±12, ±16, and ±24. A positive value indicates a left shift for `lrectwrite` and `rectcopy` and a right shift for `lrectread`; a negative value indicates a right shift for `lrectwrite` and `rectcopy` and a left shift for `lrectread`. The default value is 0.

10.4.2 Expanding Pixels

You can expand a single-bit pixel into one of two 32-bit values using PM_EXPAND with PM_C0 and PM_C1. When you set the `pixmode` value PM_EXPAND to 1, the least significant bit of a pixel's value controls the conversion of that bit into PM_C0 or PM_C1. If the bit is 0, PM_C0 is selected; if it is 1, PM_C1 is selected. For example, to convert a series of single-bit pixels (32-bit pixels whose most significant 31 bits are ignored) into blue for 0 and green for 1 in RGB mode, call:

```
pixmode(PM_EXPAND, 1);
pixmode(PM_C0, 0x00ff0000);
pixmode(PM_C1, 0x0000ff00);
```

before calling `lrectwrite`, `lrectread`, or `rectcopy`, depending on whether you want to expand pixels to be written, read, or copied. The call `pixmode`(PM_EXPAND,1) turns expansion on. The next two calls set the expansion values for the single-bit values 0 and 1, respectively.

To turn expansion off, call `pixmode`(PM_EXPAND, 0). This is the default. You can set PM_C0 and PM_C1 to any 32-bit value. In color map mode, the value of the color index is replaced by either PM_C0 or PM_C1. Their default values are 0.

PM_SHIFT can be used with PM_EXPAND to cause expansion based on a bit other than the least significant one.

10.4.3 Adding Pixels

You can add a constant signed 24-bit value to each pixel transferred by calling:

```
pixmode(PM_ADD24, value);
```

where *value* is the signed 24-bit value. This feature is most effectively used when transferring *z* data, but it also affects color data. To turn off pixel addition, call `pixmode`(PM_ADD24, 0). This is the default.

10.4.4 Pixels Destined for the *z*-Buffer

You can specify that pixels be sent to the *z*-buffer by setting:

```
pixmode(PM_ZDATA,1);
```

Unlike setting `zdraw`(TRUE), using `pixmode`(PM_ZDATA,1) treats transferred pixels as *z* data rather than color data. If you have called `zbuffer`(TRUE), the writing is conditional based on a comparison of the pixel's value with the value present in the corresponding location of the *z*-buffer. The current setting of `zfunction` determines the write condition.

To turn off this feature, call:

```
pixmode(PM_ZDATA, 0);
```

This is the default. PM_ZDATA has no effect on `lrectread`.

10.4.5 Changing Pixel Fill Directions

The default directions for reading, writing, and copying pixel rectangles are left-to-right and bottom-to-top. For example, when writing a rectangle, the first pixel is placed in the lower left-hand corner of the specified rectangle, the next at the same screen *y* value but at a screen *x* value of one greater, and so on until a line of pixels is complete. The next line is placed on top of the last until the rectangle is complete.

You can change the default reading, writing, and copying directions. Calling:

```
pixmode(PM_TTOB, 1);
```

makes the pixel transfer direction top-to-bottom.

```
pixmode(PM_RTOL, 1);
```

makes the fill direction right-to-left. For instance, you can mirror a pixel rectangle that is already in the frame buffer about a vertical line to produce a new rectangle by calling:

```
pixmode(PM_RTOL, 1);
rectcopy( ... );
```

You can set both PM_TTOB and PM_RTOL to 1 if you choose. Reset the directions to the defaults with:

```
pixmode(PM_TTOB, 0);
pixmode(PM_RTOL, 0);
```

which specify bottom-to-top and left-to-right fill directions, respectively.

These functions change the order in which pixels are filled, but do not affect the fundamental row-major ordering of pixels. That is, a horizontal line of pixels always occupies a set of contiguous words in memory, and a group of words representing one line of pixels is followed in memory by a group of words representing the next. Pixels are always arranged so that a group of contiguous words forms a horizontal line, never a vertical line. Changing fill directions does not allow interchanging a horizontal line of pixels for a vertical one.

Fill direction does not affect the location of the destination rectangle for `rectcopy`. The destination rectangle is always specified by its lower-left pixel, regardless of fill direction.

There are no restrictions on using these modes with `lrectwrite` or `lrectread`.

10.4.6 Subimages within Images

Using `pixmode` allows you to read and write pixel subrectangles to and from a larger pixel rectangle. Suppose you have a 2000 x 1500 pixel image and want to work with a 100 x 200 subimage whose origin is at (150,500) in the larger rectangle. You need to tell the GL the width of the larger rectangle with `pixmode`:

```
pixmode(PM_STRIDE, 2000);
```

PM_STRIDE specifies the number of 32-bit words per scanline of your rectangle. Next you need to compute the address of the starting pixel of your subrectangle within the large rectangle. If *p* points to the lower-left pixel of the large rectangle, then, assuming the default pixel fill directions, this address is:

```
p + (2000 * 500) + 150
```

You can then call `lrectread` or `lrectwrite`:

```
lrectread(x, y, x + 99, y + 199, p + (2000 * 500) + 150 );
lrectwrite(x, y, x + 99, y + 199, p + (2000 * 500) + 150 );
```

to read or write the subimage from or to the location (*x*,*y*) on the screen. The GL figures out where the appropriate subimage is located in CPU memory to effect the desired transfer within the larger rectangle. You can use this method to work with subrectangles of any size and offset as long as you tell the GL the width of the whole rectangle using PM_STRIDE.

The default value for PM_STRIDE is 0. PM_STRIDE has no effect on `rectcopy`.

10.4.7 Packing and Unpacking Pixel Data

You can specify a number of bits-per-pixel other than 32 for pixels in CPU memory using the `pixmode` function PM_SIZE. You can use this feature to obtain more efficient packing of pixel data in CPU memory. Setting PM_SIZE to a value other than 32 (the default value) unpacks pixels from CPU memory when written using `lrectwrite` and packs pixels into CPU memory when using `lrectread`.

If you are ignoring alpha values and are using RGB mode, you can specify a PM_SIZE of 24 and pack four RGB values into three words. If you set PM_SIZE to *n* (allowable values are 1, 4, 8, 12, 16, 24, and 32), the first pixel goes in the *n* most significant bits of the first word. The next pixel is packed adjacent to the first, so that if *n* is less than 32, its bits are placed in the next most significant bits of the first word after the first *n*. If there is not enough room in the first word for all the bits of this second pixel, the leftover bits fill the most significant bits of the following word. The second word is packed tightly in the same way, and so on for the rest of the words until the end of a line of pixels.

The next line of pixels starts in the most significant bit of a new word, even if the last pixel of the previous line did not completely fill the last word of the previous line.

For instance, for a 3 x 3 rectangle with PM_SIZE set to 12, the first pixel occupies the first 12 most significant bits of the first word, the second pixel occupies the next most significant 12 bits of the first word, and the third pixel occupies the last 8 bits of the first word and the first 4 most significant bits of the second word. The next pixel begins a new line, so it occupies the 12 most significant bits of the third word, and so on.

The packing scheme makes 8- and 16-bit packing equivalent to *char* and *short* arrays, respectively, for a line of pixels. Recall, however, that an address passed to `lrectwrite` or `lrectread` must be long-word aligned.

If PM_SIZE is not 32, pixels cannot begin on a word boundary. This might require using another `pixmode` function, PM_OFFSET, when accessing subrectangles within rectangles. Calling:

```
pixmode(PM_OFFSET, n)
```

where *n* is a value between 0 and 31, indicates that the most significant *n* bits of the first word of each scanline are to be ignored. Assume you have a subrectangle of 100 x 200. Your whole image is 1250 x 1250, the origin is at (150,500), and your pixels are packed at 24 bits-per-pixel instead of 32. In this case, you have 1250 pixels at 24 bits per pixel, or 30000 pixels in each scanline. Thus there are 30000/32 (rounded up to the nearest integer, because each scanline begins with a new word), or 938 words per scanline. To find the address of the beginning of the subrectangle, you must find the offset of the 150th pixel from the first word of a scanline in the large rectangle. This offset is (150 * 24)/32, or 111, when rounded down. But 111 words is actually (111 * 32) / 24 = (exactly) 148 pixels. The 149th pixel occupies the most significant 24 bits of the the 112th word, so the 150th pixel begins 24 bits from the beginning of the 112th word.

Therefore, to access the subimage in this example, call:

```
pixmode(PM_SIZE, 24);
pixmode(PM_STRIDE, 938);
pixmode(PM_OFFSET, 24);
```

and give the pointer:

```
p + (500 * 938) + 112
```

to `lrectread` or `lrectwrite` as the location of the first pixel in the subimage. The offset of (300 * 938) accounts for the first 300 lines of the large rectangle that must be skipped, while the offset of 112 brings the pointer to the word containing the 150th pixel in the scanline. The call to `pixmode`(PM_OFFSET, 24) effects the additional required offset of 24 bits to get to the 150th pixel itself.

Setting packing and unpacking parameters (PM_SIZE and PM_OFFSET) has no effect on `rectcopy`.

10.4.8 Order of Pixel Operations

Various `pixmode` functions can be used together. The order of calls to `pixmode` is immaterial. This is because `pixmode` functions happen in a predefined order. For `lrectwrite`, this order is:

```
unpack → shift → expand → add24 → zoom
```

For `lrectread`, the order is:

```
shift → expand → add24 → pack
```

For `rectcopy` the order is:

```
shift → expand → add24 → zoom.
```

As an example, one useful combination is to display a black and white image in RGB mode from an efficiently packed 1-bit-per-pixel encoding in memory. To do this, make the following calls:

```
pixmode(PM_SIZE, 1);
pixmode(PM_EXPAND, 1);
pixmode(PM_C0, 0x00000000);
pixmode(PM_C1, 0x00ffffff);
```

before writing the image with `lrectwrite`. You might also set PM_STRIDE and PM_OFFSET if you want to handle a subimage.

You can also read and pack a black and white image using a similar method with `lrectread`, but you would have to be certain that the black pixels had least significant bits of 0 and the white pixels least significant bits of 1, because the packing discards all but one bit. Also, when reading back the image, there would be no need for expansion. In any case, the order in which you make the `pixmode` calls is unimportant.

10.5 Old-Style Pixel Access

The subroutines in this section read and write pixels. They determine the location of the pixels on the basis of the current character position (see `cmove` and `getcpos`). They attempt to read or write up to *n* pixel values, starting from the current character position and moving along a single scanline (constant *y*) in the direction of increasing *x*. The system updates that position to the pixel that follows the last one read or written. The current character position becomes undefined if the next pixel position is greater than XMAXSCREEN. The system paints pixels from left to right and clips them to the current screenmask.

These routines do not automatically wrap from one line to the next, and they ignore zoom factors set by `rectzoom`. They are sensitive to color mode. The current color mode should be set appropriately when calling the following routines. The behavior of `readpixels` and `writepixels` is undefined in RGB mode. The behaviors of readRGB and `writeRGB` are undefined in color map mode.

writepixels()

`writepixels`(*n*, *colors*) paints a row of pixels on the screen in color map mode. *n* specifies the number of pixels to paint and *colors* specifies an array containing a color for each pixel.

writeRGB()

`writeRGB`(*n*, *red*, *green*, *blue*) paints a row of pixels on the screen in RGB mode. *n* specifies the number of pixels to paint. *red*, *green*, and *blue* specify arrays of colors for each pixel. `writeRGB` supplies a 24-bit RGB value (8 bits each for red, green, and blue) for each pixel and writes that value directly into the bitplanes.

readpixels()

`readpixels`(*n*, *colors*) reads up to *n* pixel values from the bitplanes in color map mode. it returns the number of pixels the system actually reads. The values of pixels read outside the physical screen are undefined.

readRGB()

`readRGB`(*n*, *red*, *green*, *blue*) reads up to *n* pixel values from the bitplanes in RGB mode. It returns the number of pixels the system actually reads. The values of pixels read outside the physical screen are undefined.

11. Frame Buffers and Drawing Modes

This chapter describes modes used for drawing into the various bitplane configurations that can be found on IRIS-4D Series systems. Personal IRIS owners with the minimum bitplane configuration (8 bitplanes) can skip the sections dealing with overlay and underlay bitplanes, because these are not present on the 8-bitplane Personal IRIS.

The IRIS physical frame buffer is divided into four separate Graphics Library frame buffers. Normally, you draw into the standard color frame buffer, and the data you write into it is interpreted either as RGB data (if you are in RGB mode) or as indices into a color map (if you are in color map mode). All this color data is optionally modified by a gamma correction ramp.

The color represented in the color frame buffer is not necessarily the color drawn on the screen. As the cursor moves over a pixel, the cursor's color obscures the pixel's color, and the pixel returns to its normal color after the cursor passes. Similarly, when a pop-up menu is drawn over one of your windows, the underlying colors are temporarily obscured, but reappear when the pop-up menu disappears.

This chapter discusses how you can control information that is overlaid on top of the standard pixel contents. There is also an underlay capability.

The system's physical frame buffer contains some number of bits (the exact number varies from system to system). The GL can address these bits as if they were logically separate frame buffers, or sets of bitplanes. Setting a drawing mode selects one of these GL frame buffers for access. These logical GL frame buffers are the pop-up, overlay, normal, and underlay frame buffers.

Overlay bitplanes supply additional bits of information at each pixel. You can configure the system to have from 0 up to a system-dependent maximum number of bitplanes. Whenever all the overlay bitplanes contain 0 at a pixel, the color of the pixel from the standard color bitplanes is presented on the screen. If the value stored in the overlay planes is not 0, the overlay value is looked up in a separate color table, and that color is presented instead.

Underlay bitplanes are similar in concept, in that there are extra bits for each pixel, but their values are normally ignored unless the color in the standard bitplanes is 0. In that case, the underlay color is looked up in a color map and is presented. Thus, the underlay color shows up only if there is "nothing" (the pixel value = 0) in the standard bitplanes. With 2 underlay bitplanes, there are four possible underlay colors.

Overlay bitplanes are useful for such things as menus, construction lines, rubber-banding lines, etc. Underlay planes might be used for background grids that appear wherever nothing else is drawn.

The system actually has several physical bitplanes that can be used for either overlay or underlay. Two of the available bitplanes are normally reserved for window manager use; the user is free to allocate the others among overlay bitplanes, underlay bitplanes, or neither.

The 2 bitplanes normally reserved by the window manager for pop-up menus are accessible (either by themselves using a special drawing mode, or by allocating all the available overlay and underlay bitplanes). You must be careful, however, not to conflict with the window manager's use for them, so using the reserved bitplanes is discouraged.

The cursor is handled with special cursor hardware. When the color guns scan out the screen, as they cross the square region of the screen where the cursor is to be drawn, they look at the corresponding position in the cursor mask to see which color to draw there. The cursor mask can be 1 or 2 bits deep. If the cursor mask is 0, the normal color is presented. If the mask is non-0, the mask value is looked up in a color table (again, similar to overlay) to find out which color to draw. The cursor color takes precedence over even the overlay color. As with overlays, if the cursor mask is 1 bit deep, there is only one possible color; if it is 2 bits deep, the cursor can have up to three colors.

All the same operations are available for operating on overlay or underlay bitplanes as are available for the standard bitplanes in color map mode. Rather than introduce a new set of subroutines, the color map subroutines are used and they affect the overlay and underlay bitplanes if the system is in overlay or underlay mode.

Many routines that affect the operation of the standard bitplanes should not be used while in overlay or underlay drawing mode. They include `singlebuffer` (on all except VGX systems), `doublebuffer` (on all except VGX systems), `RGBmode`, `cmode`, `zbuffer`, and `multimap`. VGX systems support double-buffered underlay and overlay.

For example, in overlay mode, `color` sets the overlay color; `getcolor` gets the current overlay color; `mapcolor` affects entries in the overlay map, and `getmcolor` reads those entries. In overlay mode, all drawing routines draw into the overlay bitplanes rather than into the standard bitplanes. The routines are similarly redefined for underlay mode. Use `drawmode` (described in the next section) to get into overlay or underlay mode.

`drawmode` puts the system into overlay or underlay mode, or back into normal mode. `drawmode (OVERDRAW)` and `drawmode (UNDERDRAW)` put the system into overlay and underlay mode. `drawmode (NORMALDRAW)` returns the system to the condition where the color subroutines refer to the standard bitplanes.

Some system resources are shared among the logical frame buffers, while in other cases the system maintains individual resources for each frame buffer. Each of the four logical frame buffers (pop-up, overlay, normal, and underlay) maintains a separate version of each of the following modes, which are modified and read back based on the current drawing mode:

`backbuffer`
`cmode`
`color` or `RGBcolor`
`doublebuffer`
`frontbuffer`
`mapcolor` (a complete separate color map)
`readsource`
`RGBmode`
`singlebuffer`
`writemask` or `RGBwritemask`

The following modes, on the other hand, affect only the operation of the normal frame buffer. You can modify these modes only while the normal frame buffer is selected (see `drawmode`).

```
acsize
blink
cyclemap
multimap
onemap
setmap
stencil
stensize
swritemask
zbuffer
zdraw
zfunction
zsource
zwritemask
```

All other modes, including matrices, viewports, graphics and character positions, lighting, and many primitive rendering options are shared among the four GL frame buffers.

11.1 Configuring Overlay and Underlay Bitplanes

To set the number of user-defined bitplanes you want to use for overlay color or underlay color, call `overlay` or `underlay`, respectively. Not all systems support overlay and underlay user-defined bitplanes simultaneously (Personal IRIS and IRIS-4D/G/GT/GTX systems support only one or the other at any one time). Call `gconfig` after `overlay` or `underlay` to activate their settings.

overlay

`overlay` sets the number of user-defined bitplanes used for overlay colors.

```
void overlay(planes)
long planes;
```

planes is the number of bitplanes you want to use for the overlay color, which varies from system to system. By default, the overlay frame buffer contains 2 bitplanes in single-buffer, color-map mode. Use `overlay` to change the number of bitplanes allocated for overlay mode; use `drawmode` to specify the overlay planes as the destination for drawing mode changes. For example, to make the overlay bitplanes operate in double-buffer mode, issue the following sequence of programming statements:

```
drawmode(OVERDRAW);
doublebuffer();
gconfig();
```

The number of available overlay bitplanes varies from system to system, as follows:

- *Personal IRIS systems*: 0 or 2 single-buffer, color-map mode overlay bitplanes. (There are no overlay bitplanes in the minimum configuration of the Personal IRIS.)
- *IRIS-4D/G/GT/GTX systems*: 0, 2, or 4 single-buffer, color-map mode overlay bitplanes. (Use of 4 is discouraged, because of interference with the window manager pop-up bitplanes.)
- *IRIS-4D/VGX systems*: 0, 2, 4, or 8 single- or double-buffer color map mode overlay bitplanes. The 4- and 8-bitplane configurations use the alpha bitplanes, which are then unavailable for use in NORMALDRAW mode. Furthermore, your system must have the alpha bitplane option for you to use 4 or 8 overlay bitplanes with an IRIS-4D/VGX system.

Call `gconfig` after you specify the overlay number. `overlay` takes effect only after `gconfig` is called, which is when all bitplane requests are resolved.

underlay

`underlay` sets the number of user-defined bitplanes used for underlay color. It is the same as `overlay` except it affects the underlay colors. The default value is 0.

```
void underlay(planes)
long planes;
```

On models that cannot support simultaneous overlay and underlay, setting `underlay` to 2 forces `overlay` to 0 and vice versa. If `underlay` is 2, there are four available colors that `mapcolor` can define. This is one more than the available number of overlay colors because of the way the precedence of the frame buffers works. If the overlay planes contain 0 at any location, the system displays the contents of the normal frame buffer at that location. If the underlay planes contain 0 at a given location, the system displays the color at index 0 of the underlay frame buffer's color map when the normal bitplanes do not obscure them.

To set the contents of the underlay frame buffer's color map, you might issue the following sequence of programming statements:

```
drawmode(UNDERDRAW);
underlay(2); /* two bitplanes, four colors */
gconfig();
mapcolor(0, 0.0, 0.0, 0.0); /* black as color 0 */
mapcolor(1, 1.0, 0.0, 0.0); /* red as color 1 */
mapcolor(2, 0.0, 1.0, 0.0); /* green as color 2 */
mapcolor(3, 0.0, 0.0, 1.0); /* blue as color 3 */
```

This loads the color map for the underlay buffers with black, red, green, and blue.

The number of available underlay bitplanes varies from system to system, as follows:

- *Personal IRIS systems*: 0 or 2 single-buffer, color-map mode underlay bitplanes. (There are no underlay bitplanes in the minimum configuration of the Personal IRIS.)

- *IRIS-4D/G/GT/GTX systems*: 0, 2, or 4 single-buffer, color map mode underlay bitplanes. (Use of 4 is discouraged, because of interference with the window manager pop-up bitplanes.)

- *IRIS-4D/VGX systems*: 0, 2, 4, or 8 single- or double-buffer color map mode underlay bitplanes. The 4- and 8-bitplane configurations use the alpha bitplanes, which are unavailable for use in NORMALDRAW mode. Furthermore, your system must have the alpha bitplane option for you to use 4 or 8 underlay bitplanes with an IRIS-4D/VGX system.

Call `gconfig` after you specify the underlay number. `underlay` takes effect only after `gconfig` is called, which is when all bitplane requests are resolved.

11.2 Drawing Modes

This chapter, and previous chapters, discuss subroutines that can change the modes of the system: `RGBmode`, `cmode`, `singlebuffer`, `doublebuffer`, `multimap`, `onemap`, `overlay`, and `underlay`. All must be followed by a call to `gconfig` to take effect.

In fact, you must call `gconfig` only after any valid set has been described. For example, to get into double-buffer RGB mode, writing into the standard bitplanes, issue the following code:

```
RGBmode();
doublebuffer();
drawmode(NORMALDRAW);
gconfig();
```

When `gconfig` is called, all the mode changes take place. There is nothing wrong with calling `gconfig` after each of the subroutines listed above, although it takes longer to execute.

By default, the system behaves as though the following subroutines were executed in initialization:

```
cmode();
gconfig();
singlebuffer();
drawmode(NORMALDRAW);
```

To change just one, for example, to go to RGB mode after initialization, use the following:

```
RGBmode();
gconfig();
```

The system remains in NORMALDRAW mode and single-buffer mode unless you explicitly change them.

drawmode

`drawmode` sets the current mode for the `color` and `writemask` routines. *mode* defines the drawing mode. Calls to `color`, `getcolor`, `getwritemask`, `writemask`, `mapcolor`, and `getmcolor` are affected by the current drawing mode. Each drawing mode has its own color and writemask. By default, the writemask enables all planes, and the color is not defined. As you switch from one drawing mode to another, the current color and writemask are saved, and the previously saved color and writemask for the new mode are restored. For example, if you are in NORMALDRAW, then switch to OVERDRAW, and then switch back to NORMALDRAW, the color and writemask that were active before you switched to OVERDRAW are automatically restored.

```
void drawmode(mode)
long mode;
```

Drawing modes include:

UNDERDRAW	Sets operations for the underlay planes
NORMALDRAW	Sets operations for the normal color index or RGB planes; also sets *z* bitplanes
OVERDRAW	Sets operations for the overlay planes
PUPDRAW	Sets operations for the pop-up menu planes (this drawing mode is maintained for compatibility only and is not recommended)
CURSORDRAW	Sets operations for the cursor planes

In cursor mode, only `mapcolor` and `getmcolor` perform a function; `color`, `getcolor`, `writemask`, and `getwritemask` are ignored.

The physical frame buffer on IRIS-4D Series systems is addressed as four sets of bitplanes: `drawmode` specifies which of these four buffers is the intended destination for the bits produced by subsequent drawing (that is, geometry) and mode commands. In addition, there is a special area reserved for cursor images (Section 11.4).

You can use Gouraud shading, i.e., `shademodel(GOURAUD)` in NORMALDRAW mode. On the Personal IRIS, when you draw polygons in the overlay and underlay bitplanes, or pop-up menus, the shading model is automatically set to FLAT.

getdrawmode

`getdrawmode` returns the current drawing mode specified by `drawmode`.

```
long getdrawmode()
```

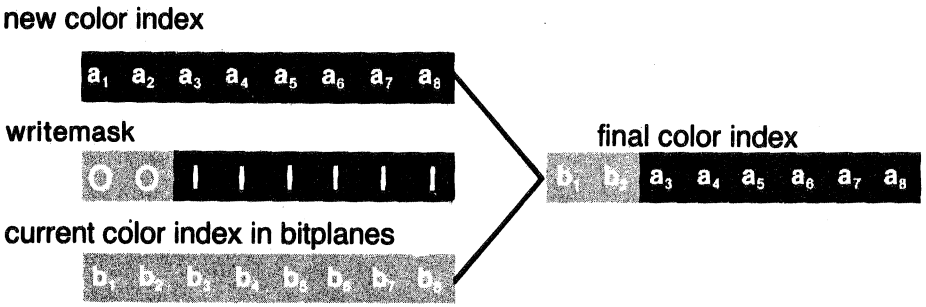
11.3 Writemasks

In all cases when the system uses color maps (the standard bitplanes in color map mode, and the overlay and underlay bitplanes), a writemask is available that can limit the drawing into the bitplanes. By default, the writemask is set up so that there are no drawing restrictions, but it is sometimes useful to limit the effects of the drawing routines. The two most common cases are to provide the equivalent of extra overlay bitplanes and to display a layered scene where the contents of the layers are independent of one another. In previous systems, overlay and underlay modes were not available; consequently, writemasks had a more significant function.

The writemask is described in terms of the standard drawing bitplanes, but exactly the same comments are true if the system is in overlay or underlay mode. This discussion assumes that only 8 of the 12 bitplanes are used, although the discussion applies equally well to different numbers.

With 8 bitplanes, the color is a number from 0 to 255, which can be represented by 8 binary bits. For example, color 68 is 01000100. Without writemask controls, if the color is set to 68, every drawing subroutine puts 01000100 into the 8 bitplanes of the affected pixels.

A writemask would restrict this. If, in the example above, the writemask were 15 (= 00001111), only the bottom 4 bits of the color are written into the bitplanes (that is, with writemask, 1 enables that bitplane for writing, and 0 disables that bitplane). If the color is 68, any pixels hit by a drawing subroutine contain wxyz0100, where wxyz are the 4 bits that were previously there. The zeros (0s) in the writemask prevent those bits from writing. The default writemask is entirely 1s, so there is no restriction (see Figure 11-1).



Writemasks determine whether or not a new value can be stored in each bitplane. A "1" in the writemask allows the system to store a new value (0 or 1) in the corresponding bitplane. A "0" prevents the system from storing a new value and the corresponding bitplane retains its current value. In this example, the values in the first and second bits (b_1 and b_2) do not change because the corresponding positions in the writemask are zero. All the other values (originally b_3,b_4, ... b_8) change to a_3,a_4, ... a_8 because the corresponding positions in the writemask are 1. Each value a_1, ... a_8 and b_1, ... b_8 is either 0 or 1.

Figure 11-1. The writemask Subroutine

As a simple example, suppose you want to draw two completely independent electronic circuits on the screen: power and ground. You want the power grid to be drawn in blue, the ground grid to be drawn in black, and short-circuits (where both power and ground appear) to be drawn in red. The background color is white.

Initialize the program as follows:

```
#define       BACKGROUND    0       /*=00*/
#define       POWER         1       /*=01*/
#define       GROUND        2       /*=10*/
#define       SHORT         3       /*=11*/

mapcolor(0, 255, 255, 255);         /*white*/
mapcolor(1, 0, 0, 255);             /*blue*/
mapcolor(2, 0, 0, 0);               /*black*/
mapcolor(3, 255, 0, 0);             /*red*/
```

Then draw all the power circuitry into bitplane 1 and the ground circuitry into bitplane 2. Where both power and ground appear, there is a 1 in both bitplanes, making color 3.

To clear the window before drawing:

```
writemask(3);
color(BACKGROUND);
clear();
```

To draw power circuitry without affecting ground circuitry:

```
writemask(1);
color(1);
<drawing subroutines>
```

To draw ground circuitry without affecting power circuitry:

```
writemask(2);
color(2);
<drawing subroutines>
```

To erase all power circuitry:

```
writemask(1);
color(0);
clear();
```

To erase all ground circuitry:

```
writemask(2);
color(0);
clear();
```

Here is a simple complete program that draws the power and ground circuitry. The interface consists of the keys `P` (draw power rectangles), `G` (draw ground rectangles), `C` (clear the window,) and `Q` (quit). To draw a rectangle, press the left mouse button at one corner, hold it down, slide to the other corner, and release it.

When you use the program, be sure to exit by typing `Q`—this resets the four lowest entries in the color map (which are used by all the windows) back to the default values. If you forget, your text will be white against a white background, and hence a bit tough to read. If this happens, type a couple of carriage returns, followed by `gclear` and another carriage return. The program `gclear` resets the color map back to the default.

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

#define BACKGROUND  0   /* = 00 */
#define POWER       1   /* = 01 */
#define GROUND      2   /* = 10 */
#define SHORT       3   /* = 11 */

long   xorg, yorg, xsize, ysize;
```

```
main()
{
    short val, drawtype;
    short x1, y1, x2, y2;

    drawtype = GROUND;
    winopen("circuit");
    mmode(MVIEWING);
    getorigin(&xorg, &yorg);
    mapcolor(0, 255, 255, 255);
    mapcolor(1, 0, 0, 255);
    mapcolor(2, 0, 0, 0);
    mapcolor(3, 255, 0, 0);
    qdevice(PKEY);       /* draw power rectangles */
    qdevice(GKEY);       /* draw ground rectangles */
    qdevice(CKEY);       /* clear screen */
    qdevice(QKEY);       /* quit */
    qdevice(LEFTMOUSE); /* mark rectangle corners */
    tie(LEFTMOUSE, MOUSEX, MOUSEY);
    color(0);
    clear();
    while (TRUE)
    switch (qread(&val)) {
        case PKEY:
           drawtype = POWER;
           break;
        case GKEY:
           drawtype = GROUND;
           break;
        case CKEY:
           clearscreen();
           break;
        case QKEY:
```

```
        /* restore default colors */
        mapcolor(0, 0, 0, 0);
        mapcolor(1, 255, 0, 0);
        mapcolor(2, 0, 255, 0);
        mapcolor(3, 255, 255, 0);
        return;
         case LEFTMOUSE:
        qread(&x1);
        qread(&y1);
        qread(&val);
        qread(&x2);
        qread(&y2);
        if (drawtype == POWER)
            powerrect(x1-xorg, y1-yorg, x2-xorg,
           y2-yorg);
        else
            groundrect(x1-xorg, y1-yorg, x2-xorg,
           y2-yorg);
        break;
    }
}

clearscreen()
{
    writemask(3);
    color(BACKGROUND);
    clear();
}

powerrect(x1, y1, x2, y2)
short x1, y1, x2, y2;
{
    writemask(1);
    color(1);
    rectfs(x1, y1, x2, y2);
}

groundrect(x1, y1, x2, y2)
short x1, y1, x2, y2;
{
    writemask(2);
    color(2);
    rectfs(x1, y1, x2, y2);
}
```

writemask

`writemask` grants write permission to available bitplanes. It protects bitplanes, in the current drawing mode, that are reserved for special uses from ordinary drawing subroutines. *wtm* is a mask with 1 bit available per bitplane. Whenever there are 1s in the writemask, the corresponding bits in the color index are written into the bitplanes. Zeros in the writemask mark bitplanes as read-only. These bitplanes are unchanged, regardless of the bits in the color. If the drawing mode is NORMALDRAW, `writemask` affects the standard bitplanes; if it is OVERDRAW, it affects the overlay bitplanes; if it is UNDERDRAW, it affects the underlay bitplanes. Use `RGBwritemask` in RGB mode.

```
void writemask(wtm)
Colorindex wtm;
```

It is very important to understand that although `writemask` allows you to protect certain bits from being overwritten, all the bits stored at any pixel are still taken as a single integer or color index value.

Consider as an example the following program:

```
#include <gl/gl.h>
main()
{
    prefposition(100, 750, 100, 750);
    winopen("Circles");
    mmode(MVIEWING);
    color(BLACK);
    clear();
    color(RED);
    circfi (200, 400, 200);
    writemask (0xFFF - 1); /* Zero the #1 bitplane */
    color (GREEN) ;
    circfi (400, 400, 200);
    writemask (0xFFF - 3); /* Zero the #1 & #2 bitplanes */
    color (BLUE);
    circfi (300, 250, 200);
    sleep (3);
    gexit ();
    exit(0);
}
```

This program draws overlapping circles. Because of the writemask calls, the overlapping colors form their compound color — that is, where the red and the green circle overlap, the shared area is yellow; where the red and the blue circle overlap, the shared area is magenta, etc.

getwritemask

`getwritemask` returns the current writemask of the current drawing mode. It is an integer with up to 12 significant bits, one for each available bitplane. Use `gRGBmask` in RGB mode.

```
long getwritemask()
```

RGBwritemask

`RGBwritemask` is the same as `writemask`, except it functions in RGB mode. *red*, *green*, and *blue* are masks for each of the three sets of bitplanes. In the same way that writemasks affect drawing in bitplanes in NORMALDRAW color map mode, separate red, green, and blue masks can be applied in NORMALDRAW RGB mode.

```
void RGBwritemask(red, green, blue)
short red, green, blue;
```

wmpack

`wmpack` is the same as `RGBwritemask`, except it accepts a single packed argument, rather than three separate masks. Bits 0 through 7 specify the red mask, 8 through 15 the green mask, 16 through 23 the blue mask, and 24 through 31 the alpha mask. For example, `wmpack`(0xff804020) has the same effect as `RGBwritemask`(0x20,0x40,0x80).

```
wmpack(pack)
unsigned long pack;
```

gRGBmask

`gRGBmask` returns the current RGB writemask as three 8-bit masks. `gRGBmask` places masks in the low order 8-bits of the locations *redm*, *greenm*, and *bluem* address. The system must be in RGB mode when this routine executes.

```
void gRGBmask(redm, greenm, bluem)
short *redm, *greenm, *bluem;
```

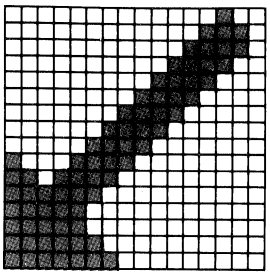
11.4 Cursor Techniques

The system supports five cursor types: a 16x16-bit cursor in one or three colors; a 32x32-bit cursor in one or three colors; and a cross-hair, one-color cursor. To specify a cursor completely, you need to specify not only its type, but its shape and color(s). In addition, every cursor has an origin, or "hot spot," and can be turned on or off. See Figure 11-2 for examples of a 16x16 one-color cursor. Three-color cursors might not be supported on all future versions of hardware. To write code that is guaranteed to be 100 percent portable, use only single-color cursors.

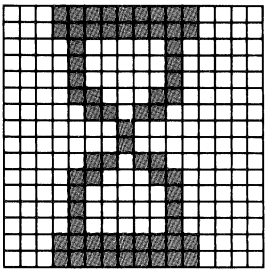
There is a default cursor, cursor number zero (0), which is an arrow pointing to the upper-left corner of the cursor glyph, and whose origin is at (0, 15), the tip of the arrow. The default cursor (number 0) cannot be redefined, and can always be used. The position of the origin of the cursor, or the cursor's hot spot, is set to the current values of the valuators that are attached to the cursor.

To define and use a new cursor, follow these steps:

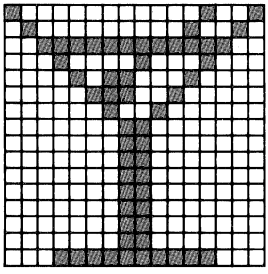
1. Set the cursor type to one of the five allowable types with `curstype`.
2. Define the cursor's shape and assign it a number with `defcursor`.
3. If necessary, define its origin (or hot spot) with `curorigin`, and its color(s) with `drawmode` and `mapcolor`.
4. Finally, the new cursor becomes the current cursor with a call to `setcursor`.



```
Cursor arrow = { 0x FE00,  0x FC00,  0x F800,  0x F800,
                 0x FC00,  0x DE00,  0x 8F00,  0x 0780,
                 0x 03C0,  0x 01E0,  0x 00F0,  0x 0078,
                 0x 003C,  0x 001E,  0x 000E,  0x 0004}
```



```
Cursor hourglass = { 0x 1FF0,  0x 1FF0,  0x 0820,  0x 0820,
                     0x 0820,  0x 0C60,  0x 06C0,  0x 0100,
                     0x 0100,  0x 06C0,  0x 0C60,  0x 0820,
                     0x 0820,  0x 0820,  0x 1FF0,  0x 1FF0}
```



```
Cursor martini = { 0x 1FF8,  0x 0180,  0x 0180,  0x 0180,
                   0x 0180,  0x 0180,  0x 0180,  0x 0180,
                   0x 0180,  0x 0240,  0x 0720,  0x 0B10,
                   0x 1088,  0x 3FFC,  0x 4022,  0x 8011}
```

A cursor is a 16X16 array of bits with the origin in the lower-left corner. The cursor is defined bottom-up, just as raster characters are defined.

Figure 11-1. Sample Cursors

If an application needs a number of different cursors, it typically defines all of them on initialization, then switches from one to another using `setcursor` (and perhaps `mapcolor`). Although they do not physically do so, cursors can be thought of as occupying 1 or 2 bitplanes of their own, which behave like overlay bitplanes as described above. A one-color cursor uses one bitplane, and a three-color cursor occupies two. Where there are 0s in the cursor's bitplane(s), the contents of the standard, overlay, and underlay bitplanes appear. In the same way that overlay colors are defined, `drawmode` and `mapcolor` define the cursor's color(s).

For a one-color cursor, first, call:

```
drawmode(CURSORDRAW)
```

followed by:

```
mapcolor(1, r, g, b)
```

For a three-color cursor, call:

```
mapcolor(1, r1, g1, b1)
mapcolor(2, r2, g2, b2)
mapcolor(3, r3, g3, b3)
```

Whenever the cursor pattern (described below) contains a 1(=01), (r1, r_1, g_1, b_1) is presented; when it is 2(=10), (r2, r_2, g_2, b_2) appears, and so on. Be sure to call `drawmode(NORMALDRAW)` after you have defined the cursor's colors.

The color of the cross-hair cursor is set by mapping color index 3.

11.4.1 Cross-Hair Cursor

The cross-hair cursor is formed with 1-pixel wide intersecting horizontal and vertical lines that extend completely across the screen. Its origin is at the intersection of the two lines. It is a one-color cursor, which always uses cursor color 3 as its color.

The cross-hair cursor is formed from a default glyph, which cannot be changed. If you assign a value to it with `defcursor`, the user-defined glyph is ignored.

curstype

`curstype` defines the current cursor type. *type* is one of C16X1, C16X2, C32X1, C32X2, and CCROSS. It is used by `defcursor` to determine the dimensions of the arrays that define the cursor's shape. C16x1 is the default value. CCROSS is a predefined cross-hair cursor, which is one pixel wide. The hot spot is at the center of the cross. Its default center is (15, 15). CCROSS uses cursor color 3.

After you call `curstype`, call `defcursor` to specify the appropriately sized array and to assign a numeric value to the cursor glyph.

```
void curstype(typ)
long type;
```

defcursor

`defcursor` defines a cursor glyph. *n* defines the cursor number, and *curs* is an array of bits of the correct size, depending on the current cursor type. The format of the array of bits is exactly the same as that for characters in a font—the 16-bit word at the lower-left is given first, then (if the cursor is 32 bits wide) the word to its right. Continue in this way to the top of the cursor for either 16 or 64 words. If the cursor is three-colored, another set of 16 or 64 words follows, again beginning at the bottom, for the second plane of the mask.

```
void defcursor(n, curs)
short n;
Cursor curs;
```

curorigin

`curorigin` sets the origin of a cursor. The origin is the point on the cursor that aligns with the current cursor valuators. The lower-left corner of the cursor has coordinates (0,0). Before calling `curorigin`, you must define the cursor with `defcursor`. *n* is an index into the cursor table created by `defcursor`. `curorigin` does not take effect until there a a call to `setcursor`.

```
void curorigin(n, xorigin, yorigin)
short n, xorigin, yorigin;
```

setcursor

`setcursor` sets the cursor characteristics. It selects a cursor glyph from among those defined with `defcursor`. *index* picks a glyph from the definition table. *color* and *wtm* are ignored. They are present for compatibility with older systems that made use of them. Set color for the cursor with `mapcolor` and `drawmode`.

```
void setcursor(index, color, wtm)
short index;
Colorindex color, wtm;
```

getcursor

`getcursor` returns the cursor characteristics. It returns two values: the cursor glyph (*index*) and a boolean value (*b*), which indicates whether the cursor is visible.

Note: *color* and *wtm* are included for compatibility with previous versions; they provide no useful information.

The default is the glyph index 0 in the cursor table, displayed with the color 1, drawn in the first available bitplane, and automatically updated on each vertical retrace.

```
void getcursor(index, color, wtm, b)
short *index;
Colorindex *color, *wtm;
Boolean *b;
```

Here is a sample program that defines a three-color 32x32 cursor in the shape of a United States flag. Unfortunately, 32x32 is small, so there is room for only 12 stars. (Note that a three-color cursor is not supported on the Personal IRIS; hence, C16X2 and C32X2 cursor types are not available on the Personal IRIS.)

```
#include "gl/gl.h"

main ()
{
    winopen("flag");
    setflag();
    color(0);
    clear();
    sleep(20);
}
```

```
setflag()
{
    static short    curs2[128] = {
    0,      0,      0,      0,
    0,      0,      0,      0,
    0,      0,      0,      0,
     0xffff, 0xffff, 0xffff, 0xffff,
     0xffff, 0xffff, 0xffff, 0xffff,
     0xffff, 0xffff, 0xffff, 0xffff,
     0xffff, 0xffff, 0xffff, 0xffff,
     0xffff, 0xffff, 0xffff, 0xffff,
     0xffff, 0xffff, 0xffff, 0xffff,
     0xffff, 0xffff, 0xffff, 0xffff,
      0,     0xffff, 0x6666, 0xffff,
     0x6666, 0xffff,      0, 0xffff,
      0,     0xffff, 0x6666, 0xffff,
     0x6666, 0xffff,      0, 0xffff,
      0,     0xffff, 0x6666, 0xffff,
     0x6666, 0xffff,      0, 0xffff,

     0,      0,      0,      0,
     0,      0,      0,      0,
     0,      0,      0,      0,
     0,      0,      0,      0,
     0xffff, 0xffff, 0xffff, 0xffff,
     0,      0,      0,      0,
     0xffff, 0xffff, 0xffff, 0xffff,
        0,      0,      0,      0,
     0xffff, 0xffff, 0xffff, 0xffff,
     0,      0,      0,      0,
     0xffff, 0xffff, 0xffff, 0xffff,
     0xffff,      0, 0xffff,      0,
     0xffff, 0xffff, 0xffff, 0xffff,
     0xffff,      0, 0xffff,      0,
     0xffff, 0xffff, 0xffff, 0xffff,
     0xffff,      0, 0xffff,      0 };
```

```
    curstype(C32X2);
    drawmode(CURSORDRAW);
    mapcolor(1,255,0,0);
    mapcolor(2,0,0,255);
    mapcolor(3,255,255,255);
    defcursor(1,curs2);
    setcursor(1,0,0);
    drawmode(NORMALDRAW);
}
```

12. Picking and Selecting

Up to now, you have learned how to define objects in world coordinates so the system can draw them on the screen. This chapter discusses the reverse process: how to determine what routines draw objects in a specified area of the screen. There are two ways to do this:

- `mapw` takes 2-D screen coordinates and identifies the corresponding 3-D world coordinates
- `pick` and `select`, which identify objects drawn in a specified 3-D area

Because the `mapw` command uses graphical objects to map between screen and world coordinates, this command is discussed in Chapter 16, "Graphical Objects." This chapter discusses picking and selecting.

12.1 Picking

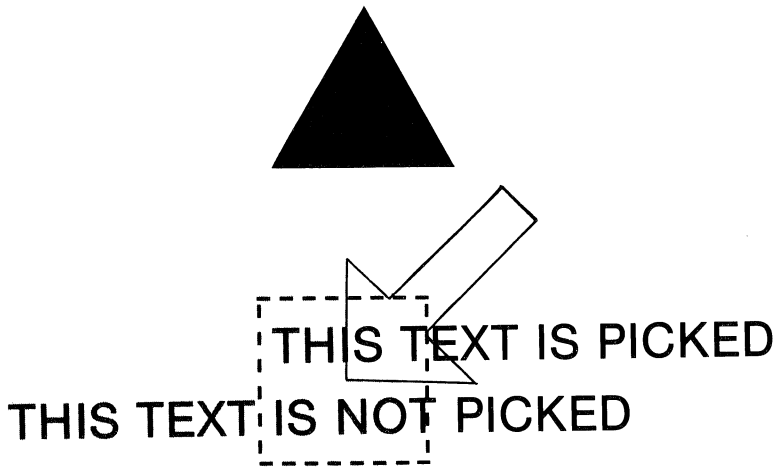
Picking mode identifies objects on the screen that appear near the cursor. To use picking effectively, your software must be structured in such a way that you can regenerate the picture on the screen whenever picking is required. When it is, set the system into picking mode, using `pick`, redraw the image on the screen, and finally, call `endpick`. The results of the pick appear in the buffer specified by `pick` and `endpick`.

While it is in picking mode, the system does not draw anything on the screen. Instead, drawing routines that would have been drawn near the cursor cause hits to be recorded in the picking buffer in a manner described below. With one exception, all the standard drawing routines cause hits, including clear, points, lines, polygons, arcs, circles, curves, and patches. Raster objects, such as character strings and pixels drawn with `charstr`, do not cause hits, but `cmov` does. Thus, to be picked, the cursor must be near the lower-left corner of the string. Note also that since `readpixels` and `readRGB` are often preceded by `cmov`, these routines can appear to cause hits. See Figure 12-1.

To identify the object(s) on the screen that caused hits, a name stack is supported. The name stack is a stack of 16-bit names whose contents are controlled by `loadname`, `pushname`, `popname`, and `initname`. In picking mode, when you issue one of the routines that alters the name stack or when you exit picking mode with `endpick` (or `endselect`, which is described in a later section), the contents of the name stack is recorded if a hit occurred since the last time the name stack was altered.

For example, suppose that your application draws three points widely spaced on the screen, and you want to find which one is close to the cursor using picking mode. Your point-drawing code (that is executed both to draw points and to redraw them in a picking operation), might look something like this:

```
ortho(<ortho parameters>);
lookat (<lookat parameters>);
translate (-x, -y, -z);
rotate(30, 'y');
translate(x, y, z);
loadname(0);
pnt(<point 0>);
loadname(1);
pnt(<point 1>);
loadname(2);
pnt(<point 2>);
```



In picking mode, you can identify the parts of an image that lie near the cursor. The cursor is shown as an arrow. The small box at the tip of the arrow is the *picking region*. The large shaded circle is picked. The text string whose origin is in the picking region is also picked. The shaded triangle and the other text string are not picked.

Figure 12-1. Picking

You must have the complete specification for drawing the picture, including any viewing and transformation routines. When this code segment is executed in picking mode, if the cursor is near point 1, the buffer returned after `endpick` contains the name 1; if it is near point 2, the buffer contains the name 2. If the cursor is not near any of the points, the system returns an empty buffer.

If hierarchical objects are drawn (for example, a car with four instances of a wheel, each wheel with five instances of a bolt, and you want to pick an individual bolt from the picture), the name stack can be used effectively. You might have one piece of code to draw each wheel that contains the sequence:

```
pushname(0);
<draw bolt 0>
popname();
pushname(1);
<draw bolt 1>
popname();
    .
    .
    .
```

The car drawing code might look like this:

```
loadname(0);
<translate>
<draw wheel>
loadname(1);
<translate>
<draw wheel>
    .
    .
    .
```

Each hit on a bolt occurs with the name stack containing two names: the first is the wheel number and the second is the bolt number on that wheel. Deeper nesting of the hierarchy is also possible.

The names reported on hits are completely application-dependent. Many drawing routines can occur between changes to the name stack, and if any of those routines cause a hit, the contents of the name stack is reported. Because the contents of the name stack is reported only when it changes, one hit is reported no matter how many of the drawing routines actually draw something near the cursor. If the application requires more accuracy than this, it must simply modify the name stack more often. In the code below, if all three points caused hits, three identical name stacks would be reported:

```
loadname(1);
pnt(-);
loadname(1);
pnt(-);
loadname(1);
pnt(-);
```

pick

`pick` puts the system in picking mode. The *numnames* argument to `pick` specifies the maximum number of values that the buffer can store. The graphical items that intersect the picking region are hits and store the contents of the name stack in *buffer*.

```
pick(buffer, numnames)
short buffer[];
long numnames;
```

12.1.1 Using the Name Stack

You maintain the name stack with `loadname`, `pushname`, `popname`, and `initnames`. To exit picking mode, use `endpick`. Each name in the name stack is 16 bits long, and you can store up to 1,000 names in a name stack. You can intersperse these routines with drawing routines, or you can insert them into object definitions. See Chapter 16, "Graphical Objects," for a discussion of objects.

loadname

`loadname` puts *name* at the top of the name stack and erases what was there before.

```
loadname(name)
short name;
```

pushname

`pushname` puts *name* at the top of the stack and pushes all the other names in the stack one level lower.

```
pushname(name)
short name;
```

Before the first `loadname()` is called, the current name is unpredictable. Calling `pushname()` before calling `loadname()` can cause unpredictable results.

popname

`popname` discards the name at the top of the stack and moves all the other names up one level.

```
popname()
```

initnames

`initnames` discards all the names in the stack and leaves the stack empty.

```
initnames()
```

endpick

`endpick` takes the system out of picking mode and returns the number of hits that occurred in the picking session. If `endpick` returns a positive number, the buffer stored all of the name lists. If it returns a negative number, the buffer was too small to store all the name lists; the magnitude of the returned number is the number of name lists that were stored.

buffer contains all of the name lists stored in picking mode, one list for each valid hit. The first value in each name list is the length of a name list. If a name stack is empty when a hit occurs, the first and only entry in the list for that hit is "0."

```
long endpick(buffer)
short buffer[];
```

12.1.2 Defining the Picking Region

Picking loads a projection matrix that makes the picking region fill the entire viewport. This picking matrix replaces the projection transformation matrix that is normally used when drawing routines are called. Therefore, you must restate the original projection transformation after `pick` to ensure the system maps the objects to be picked to the proper coordinates. If no projection transformation was originally issued, you must specify the default, `ortho2`. When the transformation routine is restated, the product of the transformation matrix and the picking matrix is placed at the top of the matrix stack. If you do not restate the projection transformation, picking does not work properly. Instead, the system typically picks every object, regardless of cursor position and pick size.

Specifying or defining the default `ortho2` parameters brings up the issue of creating a graphics window that has a one-to-one mapping between screen space (viewport) and world space (in this case, `ortho2`).

In the following example, assume a graphics window that is 4 pixels wide by 6 pixels high. This window runs from coordinates 0 to 3 in *x* and 0 to 5 in *y*. In order to set up a mapping between world space (floating point coordinates) and screen space (integer coordinates) that makes pixel (1,2) centered exactly at the point (1.0, 2.0) in the `ortho2` world space, you need to make the following two calls:

```
viewport(0, 3, 0, 5);
ortho2(-0.5, 3.5, -0.5, 5.5);
```

To understand why these values are correct, consider the *x* component. The width in *x* of this window is 4 pixels, which are integer values; it makes no sense to talk about pixel 1.3. In world coordinates, however, an *x* location of 1.3 is valid. The mapping from world to screen coordinates attempts to convert the *x* world coordinate 1.3 to the nearest whole-number pixel box it can find. Rounding off 1.3 points the GL at pixel 1. The call to `ortho2` runs between *x* values of -0.5 and 3.5 to let the rounding operation center the four *x* world space whole number values of 0.0, 1.0, 2.0, and 3.0 in the middle of each pixel in the *x* dimension. In this scheme, think of -0.5 as the extreme left-hand edge of the window, 3.5 as the extreme right-hand edge, 1.5 as the boundary between pixel 1 and pixel 2, etc. This lets you define the *x* range in `ortho2` so that, in effect, the world coordinates straddle the discrete whole number boundaries and center the whole numbers (0.0, 1.0, 2.0, 3.0) in the middle of each pixel (0, 1, 2, 3).

Extrapolate from this and assume a situation where the graphics window has been resized and you need to redefine a current `ortho2` based on the new size. To do this, use the following three statements:

```
getsize(&xsize, &ysize);
viewport(0, xsize - 1, 0, ysize - 1);
ortho2  (-0.5,(float)(xsize-0.5),
          -0.5,(float)(ysize-0.5);
```

In the call to `viewport`, you must subtract 1 from the value of *xsize* and *ysize* because they start at 0, not at 1. Likewise, in the call to ortho2, you need to start at -0.5, so you need to subtract -0.5 from *xsize* and *ysize* to create the straddling effect described earlier.

picksize

The default height and width of the picking region is 10 pixels centered at the cursor. You can change the picking region with `picksize`. *deltax* and *deltay* specify a rectangle centered at the current cursor position (the origin of the cursor glyph). (See Chapter 11, "Drawing Modes," for a discussion of cursors.)

```
picksize(deltax, deltay)
short deltax, deltay;
```

Example

The following program draws an object consisting of three shapes; then it loops, until you press the right mouse button. Each time you press the middle mouse button, the system:

1. enters pick mode
2. calls the object
3. records hits for any routines that draw into the picking region
4. prints out the contents of the picking buffer

Note: When you call an object in picking mode, the screen does not change. Because the picking matrix is recalculated only when you call `pick`, the system exits and reenters picking mode to obtain new cursor positions.

```
/***
    pick.c - example of picking
***/

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

#define BUFSIZE 50

void
drawit()
{
    color(RED);
    loadname(1);
    rectfi(20,20,100,100);
    loadname(2);
    pushname(21);
    circi(50, 500, 50);
    popname();
    pushname(22);
    circi(50, 530, 60);
    popname();
}
```

```
int
main()
{
short dev, val;
short buffer[BUFSIZE];
int hits;
int xsize, ysize;
int i;

    prefsize(600, 600);
    (void) winopen("pick");
    getsize(&xsize, &ysize);
    color(BLACK);
    clear();
    qdevice(LEFTMOUSE);
    qdevice(ESCKEY);
    for (i = 0;  i < BUFSIZE;  i++)  buffer[i] = 0;

    drawit();

    while (1)  {
    dev = qread(&val);
    switch (dev)  {
        case LEFTMOUSE:
        if (val == 0)  break;
        pick(buffer, BUFSIZE);
        ortho2(-0.5, xsize + 0.5,
            -0.5, ysize + 0.5);
        drawit(); /* no actual drawing takes place */
        hits = endpick(buffer);
```

```
        /* display hit information */
        {
            int index, items, h, i;

            printf("hit count: %d   hits: ", hits);
            index = 0;
            for (h = 0;  h < hits;  h++)  {
             items = buffer[index++];
             printf("(");
             for (i = 0;  i < items;  i++)  {
                if (i != 0)  printf(" ");
                printf("%d", buffer[index++]);
             }
             printf(") ");
        }
        printf("\n");
        }
        break;

        case ESCKEY: /* exit program */
        return 0;
        break;
        }
    }
}
```

When you run the program, five outcomes are possible for each picking session (the circles can be picked together because they overlap):

- nothing is picked = "hit count: 0 hits:"
- the square is picked = "hit count: 1 hits: (1)"
- the bottom circle is picked = "hit count: 1 hits: (2 21)"
- the top circle is picked = "hit count: 1 hits: (2 22)"
- both the top and bottom circles are picked = "hit count: 2 hits: (2 21) (2 22)"

12.2 Selecting

Selecting is a more general mechanism than picking for identifying the routines that draw to a particular region. A selecting region is a 2-D or 3-D area of world space. When `gselect` turns on selecting mode, the region represented by the current viewing matrix becomes the selecting region. You can change the selecting region at any time by issuing a new viewing transformation routine. To use selecting mode:

1. Issue a viewing transformation routine that specifies the selecting region.
2. Call `gselect`.
3. Call the objects or routines of interest.
4. Exit selecting mode and look to see what was selected.

gselect

`gselect` turns on the selection mode. `gselect` and `pick` are identical, except `gselect` allows you to create a viewing matrix in selection mode.

numnames specifies the maximum number of values that the buffer can store. Names are 16-bit numbers, which you can store on the name stack. Each drawing routine that intersects the selecting region causes the contents of the name stack to be stored in *buffer*. The name stack is used in the same way as it is in picking.

```
gselect(buffer, numnames)
short buffer[];
long numnames;
```

endselect

`endselect` turns off selecting mode. *buffer* stores any hits the drawing routines generated between `gselect` and `endselect`. Each name list represents the contents of the name stack when a routine was called that drew into the selecting region. `endselect` returns the number of name lists in *buffer*. If the number is negative, more routines drew into the selecting region than were specified by *numnames*.

```
long endselect(buffer)
short buffer[];
```

The following program uses selecting to determine if a rocket ship is colliding with a planet. The program calls a simplified version of the planet and draws a box representing the ship each time you press the left mouse button. The program prints `CRASH` and exits when the ship collides with the planet.

```
/***
    crash.c - example of selecting
***/

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

#define BUFSIZE 50
#define PLANET 109
#define SHIPWIDTH 20
#define SHIPHEIGHT 10

void
drawplanet()
{
    circfi(200, 200, 20);
}
```

```
int
main()
{
short dev, val;
short buffer[BUFSIZE];
int count, i;
float shipx, shipy, shipz;
int xorigin, yorigin;

    minsize(300, 300);
    (void) winopen("crash");
    getorigin(&xorigin, &yorigin);
    qdevice(LEFTMOUSE);
    qdevice(ESCKEY);
    color(BLACK);
    clear();

    for (i = 0;  i < BUFSIZE;  i++)  buffer[i] = 0;

    color(RED);
    drawplanet();

    while (TRUE)  {
    dev = qread(&val);
    switch (dev)  {
        case LEFTMOUSE:
        if (val)  {
            shipz = 0;
            shipx = getvaluator(MOUSEX) - xorigin;
            shipy = getvaluator(MOUSEY) - yorigin;

            color(BLUE);
            rect(shipx, shipy,
            shipx + SHIPWIDTH, shipy + SHIPHEIGHT);
```

```
            /* specify the selecting region to be a box
            surrounding the rocket ship */
            pushmatrix();
            ortho2(shipx, shipx + SHIPWIDTH,
            shipy, shipy + SHIPHEIGHT);
            initnames();
            gselect(buffer, BUFSIZE); /*enter selecting mode*/
            loadname(PLANET);
            drawplanet(); /* no actual drawing takes place */
            count = endselect(buffer); /* exit select mode */
            popmatrix();

            /* check to see if PLANET was selected */
            if ((count > 0) &&
                (buffer[0] == 1) &&
                    (buffer[1] == PLANET)  ) {
                printf("CRASH!\n");
            }
        }
        break;

        case ESCKEY:
        return 0;
        break;
        }
    }
}
```

13. Depth-Cueing

Chapter 8 discussed how to make an image appear more realistic by removing the hidden lines and surfaces from the image. This chapter discusses another technique that can make an image appear more realistic: depth-cueing—modifying object color based on depth. Two methods are presented in this chapter. The first method, *color replacement*, can be used with points, lines, polygons, and characters. The second method, *color blending*, can be used with points, lines, and polygons, but not characters. This method blends true object color with another color, where the blend ratio is determined by the depth of the object.

You can use color replacement to achieve an effect known as *depth-cueing*, which gives the appearance that objects closer to the viewer are brighter than those far from the viewer. Color blending is also useful for achieving such effects, but is better known for simulating atmospheric phenomena such as fog and smoke.

Depth-cueing is a fairly advanced topic. You do not need to read this on your first approach to the Graphics Library.

13.1 Depth-Cueing

Depth-cueing makes an image appear 3-D by replacing the color of all points, lines, and polygons with colors determined by their *z* values. First, turn on depth-cueing with the `depthcue` command. Then, specify a mapping of *z* values to color by using either the `lshaderange` or `lRGBrange` command. In color index mode, use `lshaderange` to describe a mapping from *z* values to color index values. In RGB mode, use `lRGBrange` to describe a mapping from *z* values to RGB values.

depthcue

`depthcue` turns depth-cue mode on and off. If *mode* is TRUE, all lines, points, characters, and polygons that the system draws are depth-cued. When *mode* is FALSE, depth-cue mode is off.

For depth cueing to work properly, the color map locations that `lshaderange` specifies must be loaded with a series of colors that gradually increase or decrease intensity.

```
depthcue(mode)
short mode;
```

getdcm

`getdcm` indicates whether depth-cue is on or off. TRUE (1) means depth-cue mode is on; FALSE (0) means depth-cue mode is off.

```
long getdcm()
```

lshaderange

`lshaderange` specifies the low-intensity color map index (*lowindex*) and the high-intensity color map index (*highindex*). These values are mapped to the near and far *z* values specified by *znear* and *zfar*.

The values of *znear* and *zfar* should correspond to or lie within the range of *z* values specified by `lsetdepth`. `lsetdepth` is the entire transformation range. `lshaderange` is the range of values where all of the shading will occur.

```
void lshaderange(lowindex, highindex, znear, zfar)
Colorindex lowindex, highindex;
long z1, z2;
```

The entries for the color map between the low index and the high index should reflect the appropriate sequence of intensities for the color being drawn. When a depth-cued point is drawn, its *z* value is used to determine its intensity. When a depth-cued line is drawn, the intensities of its points are linearly interpolated from the intensities of its endpoints, which are determined from their *z* values. You can achieve higher resolution if the near and far clipping planes bound the object as closely as possible. The following equation yields the color map index for a point with *z* coordinate. Note that this equation yields a nonlinear mapping when *z* is outside the range of *znear* . . . *zfar* and the color is limited to [*highindex, lowindex*]. Because depth-cued lines are linearly interpolated between endpoints, an endpoint outside the range of *z1* and *z2* can result in an undesirable image.

The valid range of *znear* and *zfar* depends on the compatibility mode specified by the call to `glcompat`. If GLC_ZRANGEMAP is 0, the value range for *znear* and *zfar* depends on the graphics hardware: the minimum is the value returned by `getgdesc`(GD_ZMIN) and the maximum is the value returned by `getgdesc`(GD_ZMAX). If GLC_ZRANGEMAP is 1, the minimum is 0 and the maximum is 0x7FFFFF.

$$color_z = \begin{cases} highindex, & \text{if } z \le z1; \\ (\frac{highindex - lowindex}{z2 - z1})(z - z1) + lowindex, & \text{if } z1 \le z \le z2; \\ lowindex, & \text{if } z2 \le z. \end{cases}$$

lRGBrange

`lRGBrange` sets the range of color indices to use for depth-cueing in RGB mode. *rmin* and *rmax* are the minimum and maximum values stored in the red bitplanes. Likewise, *gmin*, *gmax*, *bmin*, and *bmax* define the minimum and maximum values stored in the green and blue bitplanes, respectively. *znear* and *zfar* define the *z* values that are mapped linearly into the RGB range. *z* values nearer than *znear* are mapped to *rmax*, *gmax*, and *bmax*; values farther than *zfar* are mapped to *rmin*, *gmin*, and *bmin*.

```
void lRGBrange (rmin, gmin, bmin, rmax, gmax, bmax, zmin,
    zmax)
short rmin, gmin, bmin, rmax, gmax, bmax;
long zmin, zmax;
```

The following program draws a cube filled with points that rotates as you move the mouse. Because the image is drawn in depth-cue mode, the edges of the cube and the points inside the cube that are closer to the viewer are brighter than the edges and points farther away .

```
/***
    depthcue.c - RGB depth-cued cube filled with stars
***/

#include <gl/gl.h>
#include <gl/device.h>
#include <math.h>

#define NUMPOINTS 100
float Points[NUMPOINTS][3];

/* draws a cube centered at the origin */
void
drawcube()
{
typedef struct { int x, y, z; } Vector3i;
static Vector3i corner[8] = {
    {-1,-1,-1}, {-1,1,-1}, {1,-1,-1}, {1,1,-1},
    {-1,-1,1}, {-1,1,1}, {1,-1,1}, {1,1,1},
};
static int line[12][2] = {
    {0,1}, {1,3}, {3,2}, {2,0},
    {4,5}, {5,7}, {7,6}, {6,4},
    {0,4}, {1,5}, {2,6}, {3,7},
};
```

```
int i;

    for (i = 0;  i < 12;  i++)  {
    bgnline();
    v3i(corner[line[i]]);
    v3i(corner[line[i]]);
    endline();
    }
}

void
drawit()
{
int i;

    bgnpoint();
    for (i = 0;  i < NUMPOINTS;  i++)  v3f(Points[i]);
    endpoint();
    drawcube();
}

int
main()
```

```
{
float xrot, yrot;
short val;
int i;

    prefsize(800, 800);
    (void) winopen("depthcue");
    doublebuffer();
    RGBmode();
    gconfig();
    qdevice(ESCKEY);
    cpack(0);
    clear();
    swapbuffers();
    clear();
    ortho(-1.8, 1.8, -1.8, 1.8, -1.8, 1.8);

    /* set the range of z values to be used */
    lsetdepth(0, 0x7f0000);
    /* set up the mapping of z values to cyan intensity */
    lRGBrange(0, 0, 0, 0, 255, 255, 0, 0x7f0000);
    /* now z values control the intensity */
    depthcue(TRUE);

    /* generate random points */
    for (i = 0;  i < NUMPOINTS;  i++)  {
   Points[i][0] = 1.9 * (drand48() - 0.5);
   Points[i][1] = 1.9 * (drand48() - 0.5);
   Points[i][2] = 1.9 * (drand48() - 0.5);
    }
```

```
    while (TRUE)  {
    if (qtest() && (qread(&val) == ESCKEY))  break;

    xrot = (getvaluator(MOUSEY) -
        0.5 * YMAXSCREEN) / YMAXSCREEN;
    yrot = (getvaluator(MOUSEX) -
        0.5 * XMAXSCREEN) / XMAXSCREEN;
    rot(xrot * 4.0, 'x');
    rot(yrot * 4.0, 'y');
    cpack(0);
    clear();
    drawit();
    swapbuffers();
    }

    return 0;
}
```

13.2 Atmospheric Effects

A variety of special effects are now available in IRIS-4D/VGX systems by using the new `fogvertex` command. The command allows you to modify the color of an object based on the distance of the object from the view point. As its name suggests, `fogvertex` simulates atmospheric effects such as fog or smoke. It can also provide other distance-varying functions such as depth-cueing.

Consider looking down the runway at an approaching aircraft. On a clear sunny day, the aircraft is seen in full detail, limited only by your visual acuity (the resolution of your eye). On a foggy day, however, your view of the airplane is impaired and the apparent color of the plane is a combination of the fog color and the true color of the plane. As the plane approaches, though, your eye begins to detect the true color of the plane.

Such an effect can be easily simulated with the `fogvertex` command. The idea is that true color is blended with a "fog color" based on the distance of the object to the viewer to produce apparent color. "True color" is the color of an object if there were no fog, i.e., the color of an object computed after lighting, shading, and texture mapping (if any). Fog color is provided as an argument to the `fogvertex` command.

Real fog varies in density and this density is not uniform throughout a single patch of fog. Such non-uniformity is difficult to simulate quickly. Therefore, the `fogvertex` command allows the specification of a single fog density, a value that varies from 0.0 to 1.0. A fog density 0.0 is really no fog at all, i.e., the apparent color is the same as the real color. A fog density 1.0 is very thick fog, i.e., at a distance of one unit in eye coordinates, such a fog totally obscures the true color of the viewed object.

In addition to simulating fog and smoke, the `fogvertex` command can implement other simple distance varying modifications of real color. One is blending real color with "background color." The effect is that an object moving away from the viewer appears to fade into the background. This is another form of depth cueing. An advantage of this method over using `depthcue/lshaderange` is that it is independent of the color of the viewed objects, i.e., `lshaderange` must be called each time the object color changes, whereas `fogvertex` need only be called with the background color changes.

`fogvertex` take two arguments. The first is *mode*, indicating whether you are defining, enabling, or disabling fog effects. Note that defining a fog effect does not enable it. This must be done with a separate call. Values for *mode* can be:

```
void fogvertex(mode,params)
long mode;
float *params;
```

FG_DEFINE	Interpret *params* as a specification for subsequent fog density and color.
FG_ON	Enable the previously defined fog effect.
FG_OFF	Disable fog effects. This is the default.

The second argument to `fogvertex` is *params*. *params* is an array of floating point values. The first value in this array is the fog *density*. The next three are the red, green, and blue components, respectively, of the *fog color*, ranging in value from 0.0 to 1.0.

The proportion of the true color that contributes to the apparent color is called the blend factor, V_{fog}. When you enable fog effects, the blend factor is computed at the vertices of each graphics primitive. Then vertex blend factors are interpolated much like texture coordinates to determine the blend factor at other pixels covered by the graphics primitive. Finally, true color, C_p, is combined with fog color, C_f, to give apparent color, C:

$$C = C_p * V_{fog} + C_f * (1.0 - V_{fog}).$$

Computing blend factors at vertices is done according to the following formula:

$$V_{fog} = e^{(C_{fog} * density * Z_{eye})}.$$

In this equation, Z_{eye} denotes the z coordinate in eye space. When density and Z_{eye} is 1, then V_{fog} is effectively 0. (Note that when density equals 0, V_{fog} equals 1.)

`fogvertex` presumes a negative *z* orientation, i.e., the viewer is looking down the negative *z* axis from the origin. This is true for each of the GL projection matrix setting calls `perspective`, `window`, and `ortho` and when the last column of your projection matrix has the form (0.0 - 1.0).

You must be in RGB mode to use `fogvertex`.

14. Curves and Surfaces

The IRIS Graphics Library provides routines that render curved lines and surfaces. The first section in this chapter describes the interface to the Graphics Library support for non-uniform rational B-splines (NURBS). The rest of the chapter describes previous methods for drawing curves and surfaces.

Curves and surfaces are an advanced topic of the Graphics Library.

14.1 Non-Uniform Rational B-Splines (NURBS)

This section describes the routines in the Graphics Library that draw parametric non-uniform rational B-spline surfaces (NURBS) that can be trimmed with non-uniform rational B-spline curves and piecewise linear curves.

In its complete generality, the NURBS surface model can be quite complex. This chapter approaches NURBS gradually, first by examining uniform polynomial splines with no rational component, then by discussing how non-uniformity affects the splines.

The GL efficiently converts NURBS surfaces into a series of GL primitives (triangle meshes and quadrilateral strips). Therefore, as with most other Graphics Library primitives, you can transform NURBS curves and surfaces with the standard GL modeling commands. You can also use the standard lighting models when rendering NURBS curves and surfaces.

14.1.1 What Are B-Spline Curves and Surfaces?

Figure 14-1 illustrates a uniform cubic B-spline with no rational component. You define such a spline with a set of eight control points. Notice how the spline (the curve) is attracted to the control points, but does not necessarily pass through any of them.

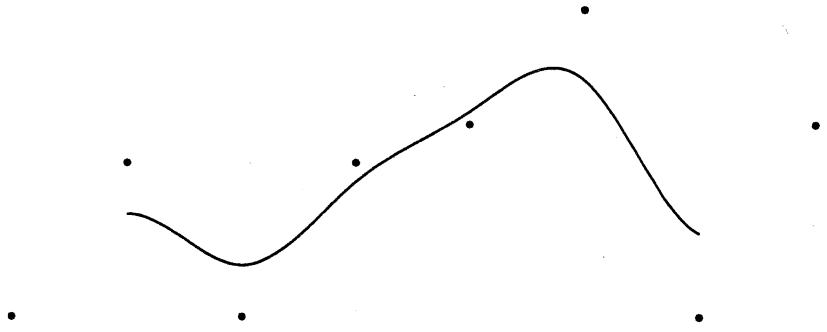


Figure 14-1. Uniform Cubic B-spline with No Rational Component

Figure 14-2 shows the effects of moving the sixth control point from the left, and the corresponding B-splines as the control points move to a series of locations. Notice that moving the control point affects only a portion of the curve near the control point. This is an important property of B-splines—the influence of the control points is local.

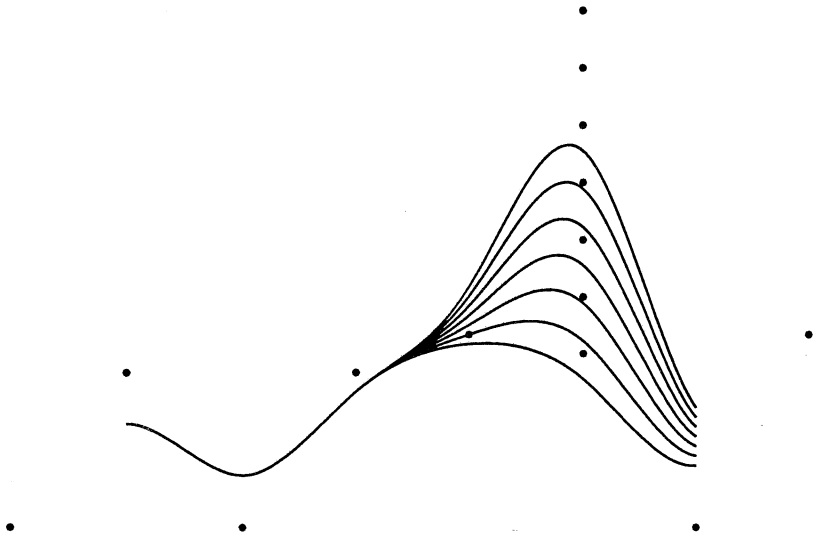


Figure 14-2. Effects of Moving a Control Point

For cubic B-splines, each small segment of the curve is controlled by the positions of four control points. In the example above, the curve is actually drawn as five small segments. The first is controlled by points 1, 2, 3, 4; the second by 2, 3, 4, 5; and so on.

The last segment is controlled by control points 5, 6, 7, and 8. When the sixth control point is moved, the only parts of the spline affected are those controlled by points 3, 4, 5, 6, points 4, 5, 6, 7, and points 5, 6, 7, 8.

In the two examples above, the control points are evenly spaced in the horizontal direction. This is not necessary. In Figure 14-3, the control points are unevenly spaced.

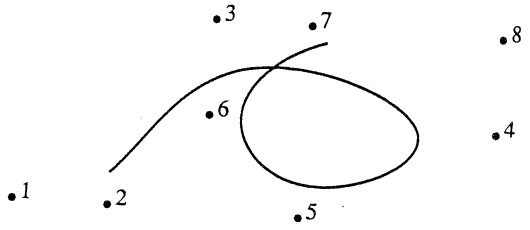


Figure 14-3. Uneven Control Point Spacing

You can use any number of control points greater than four to define a cubic B-spline. The spline is actually drawn in segments, each of which is controlled by successive sets of four control points.

14.1.2 NURBS Interface Overview

A parametric NURBS surface is the image of a rectangle in the *s*-*t* plane, called the *domain*. To describe a NURBS surface, you must specify these controlling factors:

- a set of nondecreasing knot values in both the *s* and *t* directions
- the order (which is the degree + 1) of the surface in both directions
- a rectangular set of control points

Certain dependencies exist between the surface orders, the knot counts, and the number of control points. You specify the surface orders and the knot counts to determine the number and arrangement of the control points. If *Os* and *Ot* are the surface orders in the s and t directions, and if *Ks* and *Kt* are the knot counts in those directions, then the control points must form a rectangular array of size (*Ks* - *Os*)*(*Kt* - *Ot*).

14.1.3 NURBS Surface Description

To create a NURBS surface, call `nurbssurface` within a `bgnsurface/endsurface` pair. Within this surface description, the `nurbssurface` routine lets you define three kinds of NURBS surfaces:

- A parametric surface that has shape in three dimensions. With no other modifications, this surface appears in the currently bound material and with the currently bound lights and lighting model.

- An array of color values determining the color of the geometric surface. This color has no effect when lighting is on (unless you have called `lmcolor` with an appropriate argument), just as color commands have no effect on polygons when lighting is on. Unlike the parametric surface array, the color array has no underlying geometry of its own, but is rather a function that is applied to the NURBS geometry as defined by the parametric surface geometry.

- A texture array that determines texture coordinates for the geometric surface, taking into account the current color defined by the current lighting model in conjunction with any color defined by a color `nurbssurface` call. You must call `texbind` to define a current texture for the texture coordinates to have any effect.

You must define one (and only one) geometric surface within each `bgn/endsurface` pair. You can optionally specify one color array and one texture array per parametric surface. If you do, the defined color or texture is applied to the NURBS surface in much the way that color-per-vertex or texture mapping is applied to polygons.

For color and texture arrays, the number, spacing, or sequence of the control points is completely independent of the number, spacing, or sequence of the parametric control points that define the surface's shape. To take a simple example, consider the chromaticity diagram with which you might be familiar — the diagram showing the spectral colors of a theoretical black body at various radiating temperatures. In the `nurbssurface` model, you make two calls to `nurbssurface` to create this image. The first call contains the geometric parameters, and defines the shape of the chromaticity diagram (a bilinear, or flat, surface). The second call contains an array of color values that are applied to the shape that the first call defines.

For texture mapping, the texture array passed to the `nurbssurface` call must contain an array of *s* and *t* texture coordinates that are associated with the surface geometry. For more information on defining textures, see Chapter 18, "Texture Mapping".

A parametric surface is defined by a function such as:

$$f(s, t) = (x, y, z)$$

or, for rational components:

$$f(s, t) = (x, y, z, w) = (wx, wy, wz, w)$$

You specify this mapping with a call to `nurbssurface` that defines the geometric data. A call to `nurbssurface` with color data augments this function so that it becomes:

$$f_C(s, t) = (x, y, z, r, g, b, a)$$

where r, g, b, and a give the color of the surface at (x, y, z). Specifying texture coordinates gives the function:

$$f_{tex}(s, t) = (x, y, z, s_{tex}, t_{tex})$$

Specifying both color and texture gives the most general function:

$$f_{ctex}(s, t) = (x, y, z, r, g, b, a, s_{tex}, t_{tex})$$

nurbssurface

The implementation of NURBS surfaces shown below permits orders of 8 or less in either dimension.

```
void nurbssurface (sknot_count, s_knot, tknot_count, t_knot,
    s_offset, t_offset, ctlarray, s_order,
    t_order, type)
    long sknot_count;
    double s_knot[];
    long tknot_count;
    double t_knot[];
    long s_byte_stride;
    long t_byte_stride;
    double  *ctlarray;
    long s_order;
    long t_order;
    long type;
```

The parameters to `nurbssurface` have the following meaning:

sknot_count — Indicates the number of knots in the parametric *s* direction.

tknot_count — Indicates the number of knots in the parametric *t* direction.

s_knot[] — An array of length `sknot_count`, containing the non-decreasing knot values in the parametric *s* direction.

t_knot[] — An array of length `tknot_count`, containing the non-decreasing knot values in the *t* direction.

s_order — The order of the surface in the *s* direction.

t_order — The order of the surface in the *t* direction.

type — Defines the kind of NURBS surface to be created by the call to `nurbssurface`. Can be one of several constants with the following interpretations:

`N_V3D` — Indicates that the array of control points consists of double-precision 3-D positional coordinates in non-rational (*x*, *y*, and *z*) form.

`N_V3DR` — As for N_V3D, but indicates that the array's elements are in rational (*wx*, *wy*, *wz*, and *w*) form.

`N_C4D` — Indicates that the array consists of double-precision color coordinates in four-component (R, G, B, and A) format.

`N_C4DR` — As for N_C4D, but indicates that the array's elements are in rational form.

`N_T2D` — Indicates that the array consists of double-precision texture coordinates in a two-dimensional (*s* and *t)* coordinate space.

	`N_T2DR` As for N_T2D, but indicates that the array's elements are in rational form.

You must have exactly one call to `nurbssurface` using `type N_V3D` (or `N_V3DR`, depending on whether you are using three-dimensional or rational components) inside each `bgnsurface/endsurface` loop. You can optionally include one call that specifies one of the color types (`N_C4D` or `N_C4DR`) and one that specifies one of the texture types (`N_T2D` or `N_T2DR`).

s_offset Indicates the offset (in bytes) between successive control points in the *s* direction.

t_offset Indicates the offset (in bytes) between successive control points in the *t* direction.

ctlarray An array containing control points for the NURBS surface. The coordinates must appear as triples or quadruples, depending on the value of `type` selected. That is, if you specify `type N_V3D` indicating non-rational values for the point geometry, `ctlarray` must contain elements in the form of (x, y, z) triples. For a call to `nurbssurface` that specifies `N_C4D`, that is, using non-rational color values, `ctlarray` must contain an array of (R, G, B, A) specifications in double-precision form. The values of `s_offset` and `t_offset` tell the system how many bytes make up each control point in `ctlarray`.

This interface is powerful in that the only requirement is that the *x*, *y*, *z*, and possibly *w* coordinates are placed in successive memory locations. The data can be a part of a larger data structure, or the points can form part of a larger array. For example, suppose the data appears as follows:

```
struct ptdata {
                long tag1, tag2;
                float x, y, z, w;
               }
points[5][6];
```

Set `s_offset` to `sizeof(struct ptdata)`, `t_offset` to `6*sizeof(struct ptdata)`, and `ctlarray` to `ptdata` or `&(points[0][0].x)`.

As another example, suppose that you declared the data as above, but you needed only a square of 4 by 4 control points from the middle of the array, including everything between and including `points[1][1]` and `points[4][4]`. In that case, set `s_offset` and `t_offset` as above, but set `ctlarray` to `&(points[1][1].x)`.

In both examples, *type* would be `N_V4D`, since the data includes the homogeneous *w* coordinate.

14.1.4 Trimming

A trimming curve or trimming loop defines the visible regions in a NURBS surface. You can define trim curves either by NURBS curves, using the `nurbscurve` function, or by piecewise linear curves, using the function `pwlcurve`, or by any combination of the two. In either case, the trim curve must be closed—that is, the coordinates of the first and last points of the trim curve must be identical (within the tolerance 10^{-6}). Since a NURBS curve normally does not pass through any of its control points, one way to assure a closed curve is to repeat the coordinates for a control point a number of times equal to the order of the curve—for example, quadruple the control points for a fourth-order curve if you wish to make the curve pass through that point. Another technique is to construct a knot vector that generates positional continuity of the endpoints of the curve. Trim curves can be up to order 8.

When error checking is activated (see Section 14.1.5), the software sends error messages and does not display the NURBS surfaces associated with the faulty trim data. Likewise, the end points of piecewise linear curves and NURBS curves used to form a compound trim curve must touch.

A NURBS surface is the result of a mathematical function that maps domain space to model space. You determine the visible parts of the NURBS surface by defining a *trim region.* A trim region is an area in *s-t* space defined by a closed directed loop where the interior of the region is to the left of a portion of the loop when that portion is directed upwards. The surface domain can be trimmed by many such loops, as long as they describe a consistent region. The loops can neither touch nor intersect (except at their end points, which must touch), and their orientations must also be consistent. Figure 14-4 illustrates a set of 5 loops that describe a valid trimming region. The image of the shaded portion is the trimmed NURBS surface.

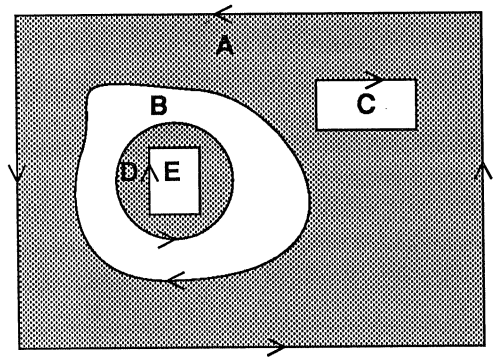


Figure 14-4. NURBS Trimming Loops

If no trimming information is provided, the trim region is effectively a square with corners at (0,0) and (1,1). If any trim loops are given, the outer loop (or loops) must be counter-clockwise. Thus, to describe a region that consists of the whole surface minus a small circle in the middle, two trim loops must be specified; one running clockwise around the circle, and another running counterclockwise around the entire *s-t* domain.

A trimming loop can be described either as piecewise linear curves (a series of *s-t* coordinates locating successive points along a path), or as NURBS curves in the *s-t* plane. A loop can be described either by a single NURBS curve or by a piecewise linear curve, or as a series of curves (of either type) joined head to tail.

In general, a sequence of calls to define a trimmed NURBS surface has a format like this:

```
bgnsurface();
    nurbssurface(. . .);
    bgntrim();
        nurbscurve(. . .);
    endtrim();
    bgntrim();
        pwlcurve(. . .);
    endtrim();
    bgntrim();
        nurbscurve(. . .);
        pwlcurve(. . .);
        nurbscurve(. . .);
    endtrim();
endsurface();
```

Each trimming loop is surrounded by a `bgntrim()` and an `endtrim()` pair. A single curve defines the first two trim loops; the third loop consists of three segments, connected head to tail. The last point of each curve segment must touch the first point of the next, and the last point of the last segment must touch the first point of the first segment. `nurbssurface()` describes the untrimmed surface, and appears before any trimming information. The trimmed surface description is bracketed by a `bgnsurface()` and `endsurface()` call.

You can use all the routines above except `getnurbsproperty()` in display lists. In this implementation, NURBS surfaces described in display lists usually run faster, since some of the display computations can be cached between display list traversals. In future implementations, this might not be true, but you can expect NURBS in display lists always to be no slower than immediate mode NURBS display.

All the arguments are passed with call-by-value semantics. This means that the system copies all values, including trim points and control points, at the time of the call. For example, if you have an array containing control points, and you define a NURBS surface in a display list using it, and then change the value in your array, the display list continues to draw the surface using the original control point values.

nurbscurve

Use `nurbscurve()` to define NURBS trim curves:

```
void nurbscurve (knot_count, knot_list, offset,
   ctlarray, order, type)
long knot_count;
double  knot_list[];
long offset;
double  *ctlarray;
long order;
long type;
```

knot_count The number of knots in the NURBS curve. As with NURBS surfaces, you define the knots and the order of the curve to obtain the control points. NURBS curves, however, being one-dimensional, have only one parametric coordinate instead of two.

knot_list An array of non-decreasing knot values.

offset The offset (in bytes) between successive curve control points in *ctlarray* (below)

ctlarray An array containing control points for the NURBS trimming curve. The coordinates must appear either as (*x,y*) pairs (requiring type N_P2D) or (*wx, wy, w*) triples (requiring type N_P2DR). The offset between successive control points in *ctlarray* is given by *offset.*

order The order or the NURBS curve. The order is equal to the degree plus one; for instance, a cubic curve is a fourth-order curve.

type A value indicating the control point type. Can be one of the following:

N_P2D non-rational coordinates (*x, y* pairs) in double-precision format

N_P2DR rational coordinates (*wx, wy, w* triples) in double-precision format

In the current implementation, you can use `nurbscurve()` only in curves of up to order 8.

The structure of the parameters is analogous to those for the `nurbssurface()` routine, except that there is only one dimension to describe.

If a single curve defines the entire trimming loop, both ends of the curve must lie at the same point and must be included in the parameter count.

When you trim a NURBS surface with a NURBS trimming curve, the software analytically calculates coordinates on the surface and their corresponding normal vectors for each point on the tessellated NURBS trimming curve.

The system displays the regions of the NURBS surface that lie to the left of a portion of the trim curve when that portion is directed upwards (upwards is defined by the direction of increasing curve parameter). For example, if your trim curve describes a counterclockwise circle, the system displays the region inside the curve. A clockwise curve causes the system to display the portion of the surface outside the trim curve (see Figure 14-4).

pwlcurve

To define a piecewise linear trimming curve, use `pwlcurve()` :

```
void pwlcurve(n, data_array, byte_size, type)
long n, byte_size, type;
double  *data_array;
```

The trim curve in the *s-t* plane is drawn by connecting each point in the `array` to the next. It is equally important to increment the trim point count as it is to duplicate the last point—in other words, although the last and first points are identical, they must be specified and counted twice.

type expects a value that indicates the point type. Currently, the only data type supported is `N_PWD`, which corresponds to pairs of *s-t* coordinates. The *byte_size* parameter specifies an offset which is used if the curve points are part of an array of larger structural elements. `pwlcurve` searches for the *n*th coordinate pair beginnint at *data_array* + *n* * *byte_size*.

14.1.5 Controlling Display Properties

The only other routines that are specifically related to the properties of NURBS surfaces are `setnurbsproperty()` and `getnurbsproperty()`. These routines allow you to set and get drawing parameters of various types.

The routine `setnurbsproperty()` changes various properties that control the rendering of NURBS curves and surfaces. The call uses this format:

```
void setnurbsproperty(long property, float value)
```

A list of properties is defined in *gl/gl.h*, and includes `N_PIXEL_TOLERANCE`, `N_ERRORCHECKING`, `N_DISPLAY`, and `N_CULLING`. Each has some reasonable default value, but can be changed to affect the accuracy of some part of the rendering. You can get the current value of any of these properties with the call:

```
getnurbsproperty(long property, float *value)
```

For maximum generality, express the value of a property in floating point. For some properties, only integer values make sense, but you must still pass them in floating point form.

The values of the properties are global to a process, and each call to `setnurbsproperty()` changes this global state.

The properties have the following meanings:

N_PIXEL_TOLERANCE	A positive floating point value that bounds the maximum length (in pixels) of an edge of any polygon generated in the tessellation of the surface.
N_ERRORCHECKING	A Boolean value that, when TRUE, instructs the GL to send NURBS-related error messages to standard error.

N_DISPLAY	An enumeration that dictates the rendering format. Possible values are N_FILL, N_OUTLINE_POLY, and N_OUTLINE_PATCH. N_FILL instructs the GL to fill all polygons generated in the tessellation of the surface. N_OUTLINE_POLY instructs the GL to outline all polygons generated. N_OUTLINE_PATCH instructs the GL to outline the boundary of all patches and trim curves.
N_CULLING	A Boolean value that, when TRUE, instructs the GL to discard before tessellation all patches that are outside the current viewport.

14.2 Old Style Curves and Surfaces

The remainder of this chapter describes the models, mathematics, and programming statements for drawing curves and surfaces that were available before the NURBS functions included in the Graphics Library for release 3.2 of the IRIX system software. These techniques and GL functions are still supported for compatibility with programs written for earlier versions of the software. Use of these statements is not recommended.

14.2.1 Overview

You draw a curve segment by specifying:

- a set of four control points
- a basis, which defines how the system uses the control points to determine the shape of the segment

You create complex curved lines by joining several curve segments end to end. The curve facility provides the means for making smooth joints between the segments.

Three-dimensional surfaces, or *patches*, are represented by a 'wireframe' of curve segments. You draw a patch by specifying:

- a set of 16 control points
- the number of curve segments to be drawn in each direction of the patch
- the two bases that define how the control points determine the shape of the patch

You can create complex surfaces by joining several patches into one large patch.

14.3 Curve Mathematics

The mathematical basis for the IRIS curve facility is the *parametric cubic curve*. The curves in most applications are too complex to be represented by a single curve segment and instead must be represented by a series of curve segments joined end to end. To create smooth joints, you must control the positions and curvatures at the endpoints of curve segments. Parametric cubic curves are the lowest-order representation of curve segments that provide continuity of position, slope, and curvature at the point where two curve segments meet.

A parametric cubic curve has the property that x, y, and z can be defined as third-order polynomials for variable t :

$$x(t) = a_x t^3 + b_x t^2 + c_x t + d_x$$

$$y(t) = a_y t^3 + b_y t^2 + c_y t + d_y$$

$$z(t) = a_z t^3 + b_z t^2 + c_z t + d_z$$

A cubic curve segment is defined over a range of values for t (usually $0<=t<=1$), and can be expressed as a vector product:

$$C(t) = at^3 + bt^2 + ct + d$$

The IRIS approximates the shape of a curve segment with a series of straight line segments. You can compute the endpoints for all the line segments by evaluating the vector product $C(t)$ for a series of t values between 0 and 1. The shape of the curve segment is determined by the coefficients of the vector product, which are stored in column vector M. These coefficients can be expressed as a function of a set of four control points. Thus, the vector product becomes

$$C(t) = T M = T (B G)$$

where G is a set of four control points, or the *geometry*, and B is a matrix called the *basis* . The basis matrix is determined from a set of constraints that express how the shape of the curve segment relates to the control points. For example, a constraint might be that one endpoint of the curve segment is located at the first control point, or the tangent vector at that endpoint lies on the line segment formed by the first two control points. When the vector product C is solved for a particular set of constraints, the coefficients of the vector product are identified as a function of four variables (the control points). Then, given four control points, you can use the vector product to generate the points on the curve segment. Three classes of cubic curves are discussed here: Bezier, Cardinal spline, and B-spline.The set of constraints that define each class is given, along with the basis matrix derived from those constraints. Section 14.4, Drawing Curves, tells how to use the basis matrices to draw curve segments.

14.3.1 Bezier Cubic Curve

A *Bezier* cubic curve segment passes through the first and fourth control points and uses the second and third points to determine the shape of the curve segment. Of the three kinds of curves, the Bezier form provides the most intuitive control over the shape of the curve. The Bezier basis matrix is derived from the following four constraints:

One endpoint of the segment is located at p1:

$$Bezier(0) = p_1$$

The other endpoint is located at p4:

$$Bezier(1) = p_4$$

The first derivative, or slope, of the segment at one endpoint is equal to this value:

$$Bezier(0)' = 3(p_2 - p_1)$$

The first derivative at the other endpoint is equal to this value:

$$Bezier(1)' = 3(p_4 - p_3)$$

Solving for these constraints yields the following equation:

$$Bezier(t) = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \end{bmatrix}$$

$$= TM_bG_b$$

You can generate all the points on the Bezier cubic curve segment from p_1 to p_4 by evaluating $Bezier(t)$ for $0 \leq t \leq 1$.

(It is more efficient, however, to construct a forward difference matrix that generates the points in a curve segment incrementally. Forward difference matrices are discussed in Section 14.2.)

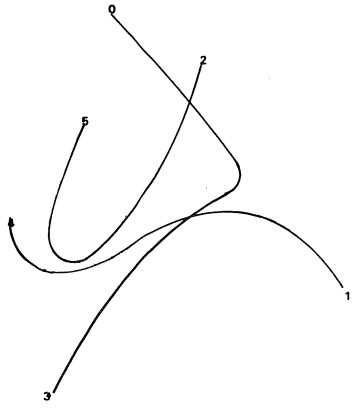
Figure 14-4 shows three Bezier curve segments. The first segment uses points 0, 1, 2, and 3 as control points. The second uses 1, 2, 3, and 4. The third uses 2, 3, 4, and 5. You can use the technique of overlapping sets of control points more effectively with the following two classes of cubic curves to create a single large curve from a series of curve segments.

14.3.2 Cardinal Spline Cubic Curve

A *spline* curve segment passes through the two interior control points and is continuous in the first derivative at the points where segments meet. The curve segment starts at $p2$ and ends at $p3$, and uses $p1$ and $p4$ to define the shape of the curve. The mathematical derivation of the Cardinal spline basis matrix can be found in James H. Clark, *Parametric Curves, Surfaces and Volumes in Computer Graphics and Computer-Aided Geometric Design*, Technical Report No. 221, Computer Systems Laboratory, Stanford University.

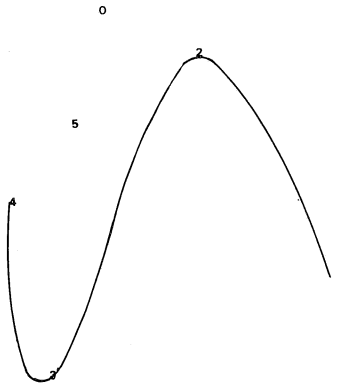
(a) Bezier

$$\begin{bmatrix} -1.0 & 3.0 & -3.0 & 1.0 \\ 3.0 & -6.0 & 3.0 & 0.0 \\ -3.0 & 3.0 & 0.0 & 0.0 \\ 1.0 & 0.0 & 0.0 & 0.0 \end{bmatrix}$$



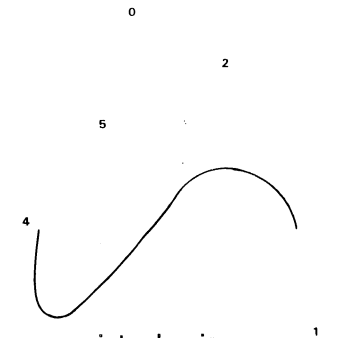
(b) Cardinal Spline

$$\begin{bmatrix} -0.5 & 1.5 & -1.5 & 0.5 \\ 1.0 & -2.5 & 2.0 & -0.5 \\ -0.5 & 0.0 & 0.5 & 0.0 \\ 0.0 & 1.0 & 0.0 & 0.0 \end{bmatrix}$$



(c) B-Spline

$$\frac{1}{6}\begin{bmatrix} -1.0 & 3.0 & -3.0 & 1.0 \\ 3.0 & -6.0 & 3.0 & 0.0 \\ -3.0 & 0.0 & 3.0 & 0.0 \\ 1.0 & 4.0 & 1.0 & 0.0 \end{bmatrix}$$



Three different curves are shown with appropriate basis matrices. With the Bezier basis matrix, three sets of overlapping control points result in three separate curve segments. With the Cardinal spline and B-spline matrices, the same overlapping sets of control points result in three joined curve segments.

Figure 14-4. Bezier, Cardinal, and B-Spline Curves

The Cardinal spline basis matrix is derived from the following four constraints:

$Cardinal(0) = p2$

$Cardinal(1) = p3$

$Cardinal(0)' = a(p3 - p1)$

$Cardinal(1)' = a(p4 - p2)$

The scalar coefficient a must be positive; it determines the length of the tangent vector at point p_2 : $tangent_2 = a(p_3 - p_1)$ and p_3 : $tangent_3 = a(p_4 - p_2)$. Solving for these constraints yields the following equation:

$$Cardinal(t) = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} \begin{bmatrix} -a & 2-a & -2+a & a \\ 2a & -3+a & 3-2a & -a \\ -a & 0 & a & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \end{bmatrix}$$

$$= TM_CG_C$$

The three joined Cardinal spline curve segments in Figure 14-1 use the same three sets of control points as the Bezier curve segments. Many different bases have Cardinal spline properties. You can derive the different bases by trying different values of a.

14.3.3 B-Spline Cubic Curve

In general, a *B-spline* curve segment does not pass through any control points, but is continuous in both the first and second derivatives at the points where segments meet. Thus, a series of joined B-spline curve segments is smoother than a series of Cardinal spline segments (see Figure 14-6).

The B-spline basis matrix is derived from the following four constraints:

$$B\text{–}spline\,(0)' = \frac{(p_3 - p_1)}{2}$$

$$B\text{–}spline\,(1)' = \frac{(p_4 - p_2)}{2}$$

$$B\text{–}spline\,(0)'' = p_1 - 2p_2 + p_3$$

$$B\text{–}spline\,(1)'' = p_2 - 2p_3 + p_4$$

Solving for these constraints yields the following equation:

$$B\text{–}spline\,(t) = \begin{bmatrix} t^3 t^2 t 1 \end{bmatrix} \frac{1}{6} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{bmatrix} \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \end{bmatrix}$$

$$= TM_B G_B$$

14.4 Drawing Curves

Drawing a curve segment on the screen involves four steps:

1. Define and name a basis matrix with `defbasis`.
2. Select a defined basis matrix as the *current basis matrix* with `curvebasis`.
3. Specify the number of line segments used to approximate each curve segment with `curveprecision`.
4. Draw the curve segment using the current basis matrix, the current curve precision, and the four control points with `crv`. `rcrv` draws a rational curve.

defbasis

`defbasis` defines and names a basis matrix to generate curves and patches. mat is saved and is associated with id. Use id in subsequent calls to `curvebasis` and `patchbasis`.

```
void defbasis(id, mat)
long id;
Matrix mat;
```

curvebasis

`curvebasis` selects a basis matrix (defined by `defbasis`) as the *current basis matrix* to draw curve segments.

```
void curvebasis(basisid)
short basisid;
```

curveprecision

`curveprecision` specifies the number of line segments used to draw a curve. Whenever `crv`, `crvn`, `rcrv`, or `rcrvn` executes, a number of straight line segments (*nsegments*) approximates each curve segment. The greater the value of *nsegments*, the smoother the curve, but the longer the drawing time.

```
void curveprecision(nsegments)
short nsegments;
```

crv

`crv` draws the curve segment using the current basis matrix, the current curve precision, and the four control points specified in the argument to `crv` .

```
void crv(geom)
Coord geom[4][3];
```

When you issue `crv`, a matrix is built from the geometry, the current basis, and the current precision:

$$M = F_{precision} M_{basis} G_{geom}$$

$$= \begin{bmatrix} \frac{6}{n^3} & 0 & 0 & 0 \\ \frac{6}{n^3} & \frac{2}{n^2} & 0 & 0 \\ \frac{1}{n^3} & \frac{1}{n^2} & \frac{1}{n} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} M_{basis} G_{geom}$$

where n = the current precision. The bottom row of the resulting transformation matrix identifies the first of n points that describe the curve. To generate the remaining points in the curve, the following algorithm is used to iterate the matrix as a forward difference matrix. The third row is added to the fourth row, the second row is added to the third row, and the first row is added to the second row. The fourth row is then output as one of the points on the curve.

```
/* This is the forward difference algorithm */
/* M is the current transformation matrix */
move (M[3][0]/M[3][3], M[3][1]/M[3][3], M[3][2]/M[3][3]);
/* iteration loop */
for (cnt = 0; cnt < iterationcount; cnt++) {
   for (i=3; i>0; i--)
      for (j=0; j<4; j++)
         M[i][j] = M[i][j] + M[i-1][j];
   draw(M[3][0]/M[3][3], M[3][1]/M[3][3], M[3][2]/M[3][3]);
}
```

Each iteration draws one line segment of the curve segment. Note that if the precision matrix on the previous page is iterated as a forward difference matrix, it generates the sequence of points:

$$(0, 0, 0, 1);\ ((\tfrac{1}{n})^3, (\tfrac{1}{n})^2, \tfrac{1}{n}, 1);\ ((\tfrac{2}{n})^3, (\tfrac{2}{n})^2, \tfrac{2}{n}, 1);\ ((\tfrac{3}{n})^3, (\tfrac{3}{n})^2, \tfrac{3}{n}, 1);\ \ldots.$$

This is the same sequence of points generated by

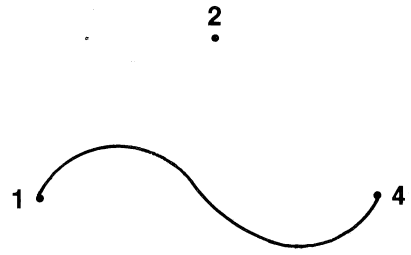
$$t = 0, \frac{1}{n}, \frac{2}{n}, \frac{3}{n}, \ldots$$

for the vector

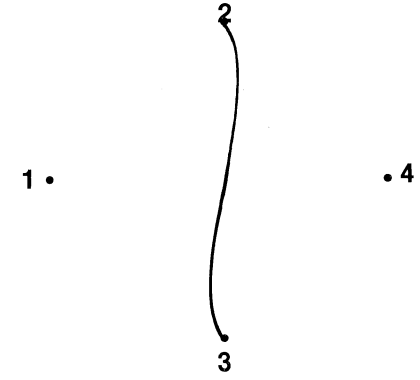
$(t^3, t\ sup\ 2, t, 1)$.

The following program draws the three curve segments in Figure 14-5. All use the same set of four control points, which is contained in *geom1*. The three basis matrix arrays (*beziermatrix*, *cardinalmatrix*, and *bsplinematrix*) contain the values discussed in the previous section. Before you call `crv` (or `rcrv`), you must define a basis and precision matrix. This is also true if the routines are compiled into an object.

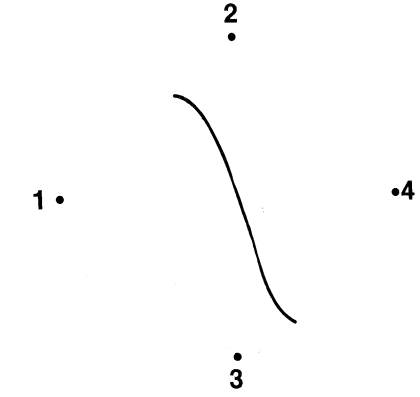
Bezier



Cardinal spline



B-spline



Each of the above curve segments uses the same set of four control points and the same precision, but a different basis matrix.

Figure 14-5. Curve Segments

Example—Curve Segments

The following program illustrates the use of curve segments in the C programming language.

```
/***
    curve1.c
***/

#include <gl/gl.h>

#define BEZIER 1
#define CARDINAL 2
#define BSPLINE 3

Matrix beziermatrix = {
    { -1, 3, -3, 1 },
    { 3, -6, 3, 0 },
    { -3, 3, 0, 0 },
    { 1, 0, 0, 0 },
};

Matrix cardinalmatrix = {
    { -0.5, 1.5, -1.5, 0.5 },
    { 1.0, -2.5, 2.0, -0.5 },
    { -0.5, 0, 0.5, 0 },
    { 0, 1, 0, 0 },
};

Matrix bsplinematrix = {
    { -1.0/6.0, 3.0/6.0, -3.0/6.0, 1.0/6.0 },
    { 3.0/6.0, -6.0/6.0, 3.0/6.0, 0 },
    { -3.0/6.0, 0, 3.0/6.0, 0 },
    { 1.0/6.0, 4.0/6.0, 1.0/6.0, 0 },
};
```

```
Coord geom1[4][3] = {
    { 100.0, 100.0, 0.0 },
    { 200.0, 200.0, 0.0 },
    { 200.0, 0.0, 0.0 },
    { 300.0, 100.0, 0.0 },
};

int
main()
{
    prefsize(400, 400);
    (void) winopen("curve1");
    color(BLACK);
    clear();

    /* define a basis matrix called BEZIER */
    defbasis(BEZIER, beziermatrix);
    /* identify the BEZIER matrix as the current basis
matrix */
    curvebasis(BEZIER);
    /* set the current precision to 20
    (the curve segment will be drawn using 20
    line segments) */
    curveprecision(20);
    color(RED);
    /* draw the curve based on the four control
    points in geom1 */
    crv(geom1);

    /* a new basis is defined */
    defbasis(CARDINAL, cardinalmatrix);
    /* the current basis is reset */
    /* note that the curveprecision does not have to
    be restated unless it is to be changed */
    curvebasis(CARDINAL);
    color(BLUE);
    /* a new curve segment is drawn */
    crv(geom1);
```

```
    /* a new basis is defined */
    defbasis(BSPLINE, bsplinematrix);
    /* the current basis is reset */
    curvebasis(BSPLINE);
    color(GREEN);
    /* a new curve segment is drawn */
    crv(geom1);

    sleep(3);
    gexit();
    exit(0);
}
```

crvn

`crvn` takes a series of *n* control points and draws a series of cubic spline or rational cubic spline curve segments using the current basis and precision; `rcrvn` draws rational splines. The control points specified in geom determine the shapes of the curve segments and are used four at a time. If the current basis is a B-spline, Cardinal spline, or basis with similar properties, the curve segments are joined end to end and appear as a single curve. Calling `crvn` has the same effect as calling a sequence of `crv` with overlapping control points (see Figure 14-1).

```
void crvn(n, geom)
long n;
Coord geom[][3];
```

When you call `crvn` with a Cardinal spline or B-spline basis, it produces a single curve. However, calling `crvn` with a Bezier basis produces several separate curve segments. As with `crv` and `rcrv`, a precision and basis must be defined before calling `crvn` or `rcrvn`. This is true even if the routines are compiled into objects. See Chapter 16 for more information on graphical objects. The following program draws the three joined curve segments in Figure 14-6 using `crvn`. *geom2* contains six control points.

Example—Joined Curve Segments

The following program demonstrates the use of joined curve segments in the C programming language.

```
/***
    curve2.c - draw a curve with a bezier,
    cardinal, and b-spline basis
***/
#include <gl/gl.h>

#define BEZIER 1
#define CARDINAL 2
#define BSPLINE 3

Matrix beziermatrix = {
    { -1, 3, -3, 1 },
    { 3, -6, 3, 0 },
    { -3, 3, 0, 0 },
    { 1, 0, 0, 0 },
};

Matrix cardinalmatrix = {
    { -0.5, 1.5, -1.5, 0.5 },
    { 1.0, -2.5, 2.0, -0.5 },
    { -0.5, 0, 0.5, 0 },
    { 0, 1, 0, 0 },
};

Matrix bsplinematrix = {
    { -1.0/6.0, 3.0/6.0, -3.0/6.0, 1.0/6.0 },
    { 3.0/6.0, -6.0/6.0, 3.0/6.0, 0 },
    { -3.0/6.0, 0, 3.0/6.0, 0 },
    { 1.0/6.0, 4.0/6.0, 1.0/6.0, 0 },
};
```

```
Coord geom2[6][3] = {
    { 150.0, 400.0, 0.0 },
    { 350.0, 100.0, 0.0 },
    { 200.0, 350.0, 0.0 },
    { 50.0, 0.0, 0.0 },
    { 0.0, 200.0, 0.0 },
    { 100.0, 300.0, 0.0 },
};

int
main()
{
    prefsize(500, 500);
    (void) winopen("curve2");
    color(BLACK);
    clear();

    /* define bezier, cardinal, and b-spline bases */
    defbasis(BEZIER, beziermatrix);
    defbasis(CARDINAL, cardinalmatrix);
    defbasis(BSPLINE, bsplinematrix);

    curveprecision(20); /* the precision is set to 20 */

    /* the Bezier matrix becomes the current basis */
    curvebasis(BEZIER);
    color(RED);
    /* the crvn command called with a bezier
    basis causes three separate curve
    segments to be drawn */
    crvn(6, geom2);

    /* the cardinal basis becomes the current basis */
    curvebasis(CARDINAL);
    color(GREEN);
    /* the crvn command called with a cardinal
    spline basis causes a smooth curve to be drawn */
    crvn(6, geom2);
```

```
    /* the b-spline basis becomes the
    current basis */
    curvebasis(BSPLINE);
    color(BLUE);
    /* the crvn command called with a b-spline
    basis causes the smoothest curve to be drawn */
    crvn(6, geom2);

    sleep(10);
    return 0;
}
```

curveit

The iteration loop of the forward difference algorithm is implemented in the Geometry Pipeline. `curveit` provides direct access to this facility, making it possible to generate a curve directly from a forward difference matrix. `curveit` iterates the current matrix (the one on top of the matrix stack) *iterationcount* times. Each iteration draws one of the line segments that approximate the curve. `curveit` does not execute the initial `move` in the forward difference algorithm. A `move(0.0,0.0,0.0)` must precede `curveit` so that the correct first point is generated from the forward difference matrix.

```
void curveit(iterationcount)
short iterationcount;
```

Example—Bezier Curve

The following program demonstrates the use of the Bezier curve in the C programming language.

```
/***
    curve3.c - example of curveit()
***/
#include <gl/gl.h>

#define BEZIER 1

Matrix beziermatrix = {
    { -1, 3, -3, 1 },
    { 3, -6, 3, 0 },
    { -3, 3, 0, 0 },
    { 1, 0, 0, 0 },
};
```

```
Matrix geom1[4][3] = {
    { 100.0, 100.0, 0.0, 1.0 },
    { 200.0, 200.0, 0.0, 1.0 },
    { 200.0, 0.0, 0.0, 1.0 },
    { 300.0, 100.0, 0.0, 1.0 },
};

Matrix precisionmatrix = {
    { 6.0/8000.0, 0, 0, 0 },
    { 6.0/8000.0, 2.0/400.0, 0, 0 },
    { 1.0/8000.0, 1.0/400.0, 1/20.0, 0 },
    { 0, 0, 0, 1 },
};

int
main()
{
    prefsize(500, 500);
    (void) winopen("curve3");
    color(BLACK);
    clear();

pushmatrix(); /* the current transformation (ortho2)
        matrix on the matrix stack is saved */
multmatrix(geom1); /* the product of the current
        transformation matrix and the matrix containing the
        control points becomes the new current
        transformation matrix */
multmatrix(beziermatrix); /* the product of the
        basis matrix and the current transformation matrix
        becomes the new current transformation matrix */
multmatrix(precisionmatrix); /* the product of the
        precision matrix and the current transformation
        matrix becomes the new current transformation
        matrix */
```

```
color(RED);
    move(0.0, 0.0, 0.0); /* this command must be issued
        so that the correct first point is generated by the
        curveit command */
curveit(20); /* a curve consisting of 20 line segments
        is drawn */
popmatrix(); /* the original transformation matrix
        is restored */

    sleep(3);
    gexit();
    exit(0);

}
```

14.4.1 Rational Curves

Cubic splines have been the focus of discussion. Cubic splines are splines whose x, y, and z coordinates can be expressed as a cubic polynomial in t.

The IRIS graphics hardware actually works in homogeneous coordinates x, y, z, and w, where 3-D coordinates are given by xw, yw, and zw. w is normally the constant 1, so the homogeneous character of the system is hidden.

In fact, the w coordinate can also be expressed as a cubic function of t, so that the 3-D coordinates of points along the curve are given as a quotient of two cubic polynomials. The only constraint is that the denominator for all three coordinates must be the same. When w is not the constant 1, but some cubic polynomial function of t, the curves generated are usually called rational cubic splines.

A circle is a useful example. There is no cubic spline that exactly matches any short segment of a circle, but if x, y, z, and w are defined as:

$$x(t) = t^2 - 1$$

$$y(t) = 2t$$

$$z(t) = 0$$

$$w(t) = t^2 + 1$$

the real coordinates

$$(x/w, y/w, z/w) = \left(\frac{t^2-1}{t^2+1}, \frac{2t}{t^2+1}, 0\right)$$

all lie on the circle with center at *(0,0,0)* in the *x*-*y* plane with radius *1* (exactly). All the conic sections (ellipses, hyperbolas, parabolas) can be similarly defined.

For rational splines, the basis definitions are identical to those for cubic splines as are the precision specifications. The only difference is that the geometry matrix must be specified in four-dimensional homogeneous coordinates. This is done with `rcrv`:

```
rcrv(geom)
coord geom[4][4];
```

`rcrv` is exactly analogous to `crv` except that w coordinates are included in the control point definitions.

rcrv

`rcrv` draws a rational curve segment using the current basis matrix, the current curve precision, and the four control points specified in its argument.

```
void rcrv(geom)
Coord geom[4] [4];
```

rcrvn

`rcrvn` takes a series of *n* control points and draws a series of rational cubic spline curve segments using the current basis and precision. The control points specified in geom determine the shapes of the curve segments and are used four at a time.

```
void rcrvn(n, geom)
long n;
Coord geom[][4];
```

14.5 Drawing Surfaces

The method for drawing surfaces is similar to that for drawing curves. A *surface patch* appears on the screen as a 'wireframe' of curve segments. A set of user-defined control points determines the shape of the patch. You can create a complex surface consisting of several joined patches by using overlapping sets of control points and the *B-spline* and *Cardinal* spline curve bases discussed in Section 14.2.

The mathematical basis for the IRIS surface facility is the *parametric bicubic surface* . Bicubic surfaces can provide continuity of position, slope, and curvature at the points where two patches meet. The points on a bicubic surface are defined by parametric equations for *x*, *y*, and *z*.

The parametric equation for *x* is:

$$x(u,v) = a_{11}u^3v^3+a_{12}u^3v^2+a_{13}u^3v+a_{14}u^3$$
$$+a_{21}u^2v^3+a_{22}u^2v_2+a_{23}u^2v+a_{24}u^2$$
$$+a_{31}uv^3+a_{32}uv^2+a_{33}uv+a_{34}u$$
$$+a_{41}v^3+a_{42}v^2+a_{43}v+a_{44}$$

(The equations for *y* and z are similar.) The points on a bicubic patch are defined by varying the parameters *u* and *v* from 0 to 1. If one parameter is held constant and the other is varied from 0 and 1, the result is a cubic curve. Thus, you can create a wireframe patch by holding *u* constant at several values and using the IRIS curve facility to draw curve segments in one direction, and then doing the same for *v* in the other direction.

To draw a surface patch, follow these steps:

1. The appropriate curve bases are defined using `defbasis` (see Section 14.1). A Bezier basis provides intuitive control over the shape of the patch. The Cardinal spline and B-spline bases allow smooth joints to be created between patches.

2. You must specify a basis for each of the directions in the patch, *u* and *v*, must be specified with `patchbasis` . Note that the *u* basis and the *v* basis do not have to be the same.

3. Use `patchcurves` to specify the number of curve segments to be drawn in each direction . A different number of curve segments can be drawn in each direction.

4. Use `patchprecision` to specify the precisions for the curve segments in each direction. The precision is the minimum number of line segments approximating each curve segment and can be different for each direction. The actual number of line segments is a multiple of the number of curve segments being drawn in the opposing direction. This guarantees the *u* and *v* curve segments that form the wireframe actually intersect.

5. Use `patch` to draw the surface. The arguments to`patch` contain the 16 control points that govern the shape of the patch. *geomx* is a 4x4 matrix containing the *x* coordinates of the 16 control points; *geomy* contains the *y* coordinates; *geomz* contains the *z* coordinates. The curve segments in the patch are drawn using the current linestyle, linewidth, color, and writemask.

`rpatch` is the same as `patch`, except it draws a rational surface patch.

patchbasis

`patchbasis` sets the current basis matrices (defined by `defbasis`) for the *u* and *v* parametric directions of a surface patch. `patch` uses the current *u* and *v* when it executes.

```
void patchbasis(uid, vid)
long uid, vid;
```

patchcurves

`patchcurves` sets the number of curves used to represent a patch. It sets the current number of *u* and *v* curves that represent a patch as a wire frame.

```
void patchcurves(ucurves, vcurves)
long ucurves, vcurves;
```

patchprecision

`patchprecision` sets the precision at which curves defining a wire frame patch are drawn. The *u* and *v* directions for a patch specify the precisions independently. Patch precisions are similar to curve precisions--they specify the minimum number of line segments used to draw a patch.

```
void patchprecision(usegments, vsegments)
long usegments, vsegments;
```

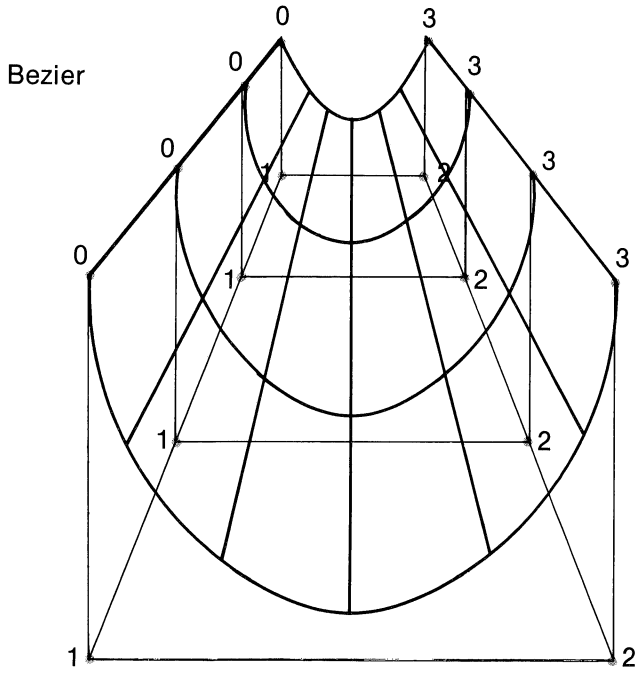
patch, rpatch

`patch` and `rpatch` draw a surface patch using the current `patchbasis`, `patchprecision`, and `patchcurves`. `rpatch` draws a rational surface patch. The control points *geomx*, *geomy*, and *geomz* determine the shape of the patch. geomw specifies the rational component of the patch to `rpatch`.

```
void patch(geomx, geomy, geomz)
Matrix geomx, geomy, geomz;

void rpatch(geomx, geomy, geomz, geomw)
Matrix geomx, geomy, geomz, geomw;
```

Figure 14-6 shows the same number of curve segments and the same precisions, but different basis matrices. All three use the same set of 16 control points.

The following program draws three surface patches similar to those shown in Figure 14-6.



Each patch is drawn using the same set of 16 control points, the same number of curve segments, the same precisions, but different basis matrices.

Figure 14-6. Surface Patches

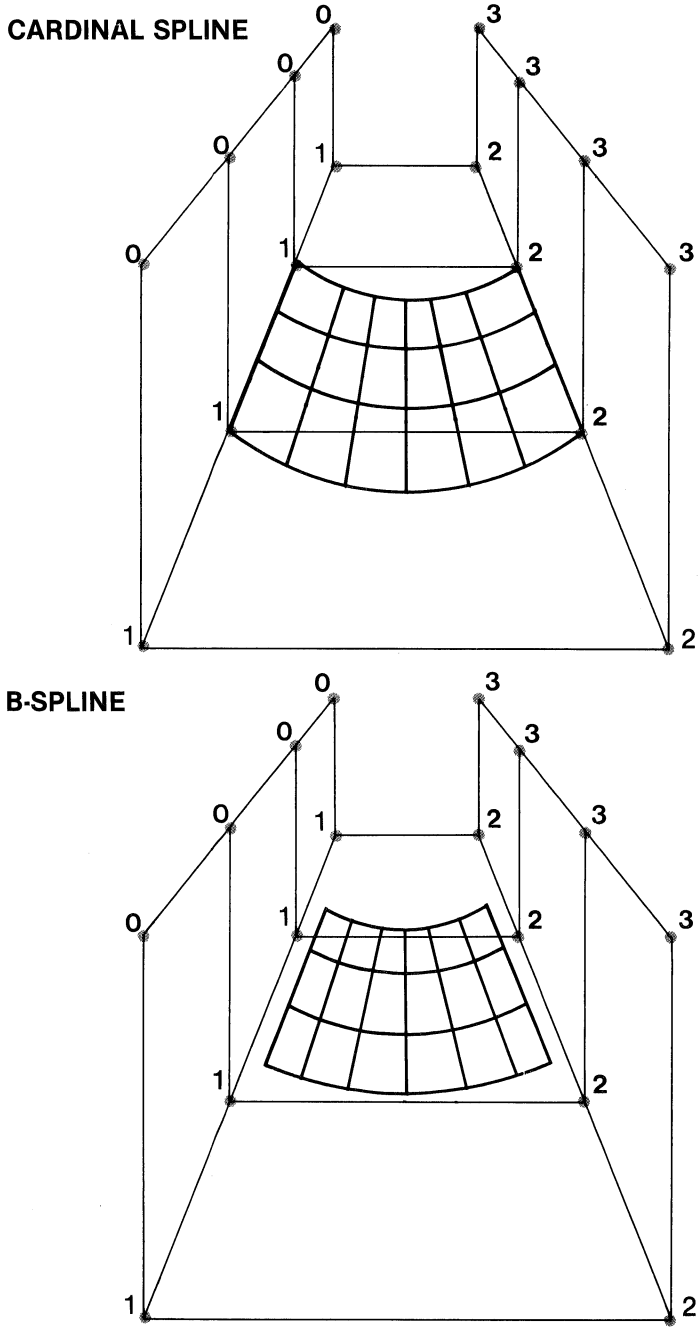


Figure 14-6. Surface Patches (continued)

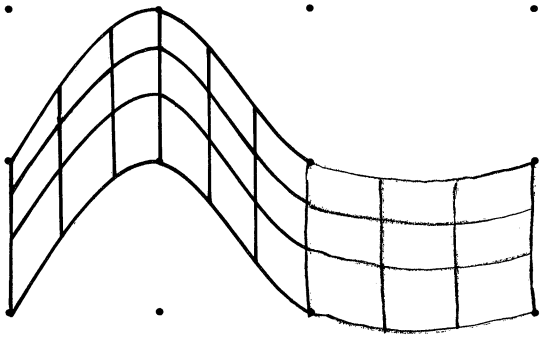
```
/***
    patch.c - draw a patch with a bezier,
    cardinal, and b-spline basis
***/

#include <gl.h>

#define BEZIER 1
#define CARDINAL 2
#define BSPLINE 3

Matrix beziermatrix = {
    { -1, 3, -3, 1 },
    { 3, -6, 3, 0 },
    { -3, 3, 0, 0 },
    { 1, 0, 0, 0 },
};

Matrix cardinalmatrix = {
    { -0.5, 1.5, -1.5, 0.5 },
    { 1.0, -2.5, 2.0, -0.5 },
    { -0.5, 0.0, 0.5, 0.0 },
    { 0.0, 1.0, 0.0, 0.0 },
};

Matrix bsplinematrix = {
    { -1.0/6.0, 3.0/6.0, -3.0/6.0, 1.0/6.0 },
    { 3.0/6.0, -6.0/6.0, 3.0/6.0, 0.0 },
    { -3.0/6.0, 0.0, 3.0/6.0, 0.0 },
    { 1.0/6.0, 4.0/6.0, 1.0/6.0, 0.0 },
};
```

```
Matrix geomx[4][4] = {
    { 0.0, 100.0, 200.0, 300.0 },
    { 0.0, 100.0, 200.0, 300.0 },
    { 700.0, 600.0, 500.0, 400.0 },
    { 700.0, 600.0, 500.0, 400.0 },
};

Matrix geomy[4][4] = {
    { 400.0, 500.0, 600.0, 700.0 },
    { 0.0, 100.0, 200.0, 300.0 },
    { 0.0, 100.0, 200.0, 300.0 },
    { 400.0, 500.0, 600.0, 700.0 },
};

Matrix geomz[4][4] = {
    { 100.0, 200.0, 300.0, 400.0 },
    { 100.0, 200.0, 300.0, 400.0 },
    { 100.0, 200.0, 300.0, 400.0 },
    { 100.0, 200.0, 300.0, 400.0 },
};

int
main()
{
int xsize, ysize;

    prefsize(700, 700);
    winopen("patch");
    getsize(&xsize, &ysize);
    RGBmode();
    gconfig();
    cpack(0);
    clear();
```

```
ortho(0.0, (float) xsize,
0.0, (float) ysize,
(float) xsize, -(float) xsize  );

/* define a basis matrix called BEZIER */
defbasis(BEZIER, beziermatrix);
/* define a basis matrix called CARDINAL */
defbasis(CARDINAL, cardinalmatrix);
/* define a basis matrix called BSPLINE */
defbasis(BSPLINE, bsplinematrix);

/* seven curve segments will be drawn in
   the u direction and four in the v direction */
patchcurves(4, 7);
/* the curve segments in the u direction
   will consist of 20 line segments (the lowest
   multiple of vcurves greater than usegments)
   and the curve segments in the v direction will
   consist of 21 line segments (the lowest
   multiple of ucurves greater than vsegments) */
patchprecision(20, 20);

/* a Bezier basis will be used for
both directions in the first patch */
patchbasis(BEZIER, BEZIER);
cpack(0xff); /* red */
wmpack(0xff);
/* the patch is drawn based on
the sixteen specified control points */
patch(geomx, geomy, geomz);

/* the bases for both directions are reset */
patchbasis(CARDINAL, CARDINAL);
```

```
    cpack(0xff00); /* green */
    wmpack(0xff00);
    /* another patch is drawn using
       the same control points but a different basis */
    patch(geomx, geomy, geomz);

    /* the bases for both directions are reset again */
    patchbasis(BSPLINE, BSPLINE);
    cpack(0xff0000); /* blue */
    wmpack(0xff0000);
    patch(geomx, geomy, geomz);

    sleep(3);
    gexit();
    exit(0);
}
```

You can join patches together to create a more complex surface by using the Cardinal spline or B-spline bases and overlapping sets of control points. The surface in Figure 14-7 consists of three joined patches and was drawn using a B-spline basis.



The above surface consists of three joined patches. The patches are drawn with overlapping sets of control points, using a Cardinal spline basis matrix.

Figure 14-7. Joined Patches

15. Antialiasing

This chapter discusses methods that make objects drawn on a discrete device—the display screen—appear smooth. The problem of smooth, or antialiased, scan conversion is discussed in Section 15.1. A prerequisite for accurate scan conversion of points, lines, and polygons is ensuring that their vertices are projected to the screen with subpixel precision. The `subpixel` routine is also discussed in Section 15.1. Some of the techniques for antialiasing RGB images use a pixel-filling technique called *blending*, which is discussed in Section 15.2.

Techniques for quickly generating antialiased points, lines, and polygons are described in Section 15.3. These techniques are fast enough to allow interactive image generation, but are sometimes difficult to use, especially in the case of polygons. A more general technique is described in Section 15.4. This technique, called *accumulation* , is an iterative process that converges on a very accurate image. It easily handles all combinations of points, lines, and polygons, but it cannot typically be used to generate an interactive image.

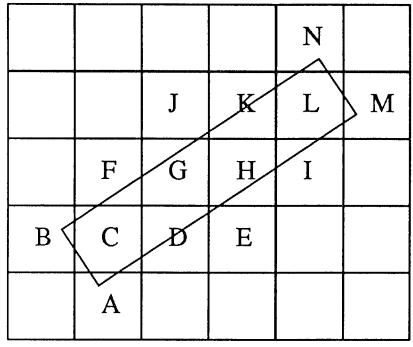


Figure 15-1. Antialiased Line

15.1 Accurate sampling

For reasons of both history and performance, the default points, lines, and polygons drawn by an IRIS-4D graphics system have jagged edges, and move from frame to frame in discrete jumps. Lines appear jagged, for example, because the true mathematical line is approximated by a series of points that are forced to lie on the pixel grid. Except in a few special cases (horizontal, vertical, and 45-degree lines) many of the approximating pixels are not on the mathematical line. Near-horizontal and near-vertical lines appear especially jagged, because their pixel approximations are a sequence of exactly horizontal or vertical segments, each offset one pixel from the next. The following program illustrates the problem:

```
/*
 * Drag a color map aliased line with the cursor.
 */

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

#define WINSIZE  400

Device devs[2] = {MOUSEX, MOUSEY};
float orig[2] = {100.,100.};

main()
{
    short val, vals[2];
    long xorg, yorg;
    float vert[2];

    presize (WINSIZE, WINSIZE);
    winopen("jagged");
    mmode (MVIEWING);
    ortho2 (-0.5, WINSIZE-0.5, -0.5, WINSIZE-0.5);
    doublebuffer();
    gconfig();
    qdevice (ESCKEY);
    getorigin (&xorg, &yorg);
```

```
    while (!(qtest() && qread(&xval) == ESCKEY && val == 0 )) {
        color(BLACK);
        clear();
        getdev (2,devs,vals);
        vert[0] = vals[0] - xorg;
        vert[1] = vals[1] - yorg;
        color(WHITE);
        bgnline();
            v2f(orig);
            v2f(vert);
        endline();
        swapbuffers();
    }
    qexit();
    return 0;
}
```

This example draws a line from the point (100,100) to the current cursor position. Move the cursor around, and notice how jagged the line appears, especially when it is nearly horizontal or nearly vertical. Even at angles far from vertical or horizontal there is some jaggedness, although it is not as noticeable.

The jaggedness that you see is called *aliasing*, and techniques to eliminate or reduce aliasing are called *antialiasing*. The line you see on the screen is aliased because it is composed of discrete pixels, each set either to the color of the line or to the background color. A much better approximation can be developed by considering the exact mathematical line to be the center line of a rectangle of width one, extending the full length of the line. A correct, antialiased sampling of this rectangle onto the pixel grid takes into account the fraction of each pixel that is obscured by the rectangle, rather than simply selecting the pixels most obscured by it. Each pixel is then set to a color between the color of the line and the color of the background, based on the fraction of the pixel that is obscured, or covered, by the line's rectangle.

To correctly sample a point, line, or polygon, the fraction of every pixel covered by the exact projection of the primitive must be computed, and that fraction must be used to determine the color of the resulting pixel. Because mathematical points and lines have no area, and therefore cannot be sampled, it is necessary to define a geometry to be used for their sampling. Points are best thought of as circles of diameter one, centered around the exact mathematical point. Lines are rectangles of width one, centered around the exact mathematical line.

subpixel

In addition to poor sampling, there is another problem with the default operation of IRIS-4D rendering. Vertices, after they have been transformed and projected to screen coordinates, are rounded to the nearest pixel center, rather than being treated with full precision. Points, lines, and polygons that are accurately sampled based on the perturbed vertices often look good in static images, but show motion artifacts when drawn in rapid animations. This is because primitives drawn with non-subpixel-positioned vertices move in discrete steps, rather than continuously. In some cases non-subpixel-positioned vertices also reduce the quality of static images. An example is a smooth curve drawn correctly sampled, but with non-subpixel-positioned vertices. This curve appears to wiggle slightly, because its component lines have been forced to end at pixel centers.

```
void subpixel(b)
Boolean b;
```

`subpixel` controls the placement of point, line, and polygon vertices in screen coordinates. By default, `subpixel` is FALSE, causing vertices to be snapped to the center of the nearest pixel after they have been transformed to screen coordinates. When `subpixel` is TRUE, vertices are positioned with at least 3 bits of fractional precision, and typically many more, up to 10 bits.

In addition to its effect on vertex position, `subpixel` also modifies the scan conversion of lines. Specifically, non-subpixel positioned lines are drawn *closed*, meaning that connected line segments both draw the pixel at their shared vertex. Subpixel positioned lines are drawn *half open*, meaning that connected line segments share no pixels. Thus subpixel positioned lines produce better results when `logicop` or `blendfunction` are used, but will produce different, possibly undesirable results in 2–D applications, where the endpoints of lines have been carefully placed.

For example, using the standard 2–D projection:

```
ortho2(left-0.5, right+0.5, bottom-0.5,top+0.5);
viewport (left,right,bottom,top);
```

subpixel positioned lines match non-subpixel positioned lines pixel for pixel, excapt that they omit either the right-most or top-most pixel. Thus the non-subpixel positioned line drawn from (0,0) to (0,2) fills pixels (0,0), (0,1), and (0,2), while the subpixel positioned line drawn between the same coordinates only fills pixels (0,0) and (0,1).

`subpixel` is not supported by all models for all primitives. Refer to the man page for details.

In most cases, drawing performance of the IRIS-4D VGX model is increased substantially when `subpixel` is true.

15.2 Blending

Not all systems support blending. See the man page for *blendfuction* for details.

By default, IRIS-4D Series systems draw pixels by replacing the pixel color value in the frame buffer with the incoming pixel color. When operating in RGB mode, however, it is possible to replace the color components of the frame buffer (destination) pixel with values that are a function of both the incoming (source) pixel color components and of the current frame buffer color components. This operation is called *blending.*

The antialiasing techniqes described in Section 15.3 require blending when operating in RGB mode. Blending has other uses, including drawing transparent objects, and compositing images. Blending is specified with the `blendfunction` command.

```
void blendfunction (sfactr,dfactr)
long sfactr, dfactr;
```

blendfunction arguments `sfactr` and `dfactr` specify how frame buffer (destination) pixels are computed, based on the incoming (source) pixel values and the current frame buffer values. By default, `sfactr` is set to BF_ONE and `dfactr` is set to BF_ZERO, resulting in simple replacement of the frame buffer color components:

```
Rdestination = Rsource
Gdestination = Gsource
Bdestination = Bsource
Adestination = Asource
```

When `blendfunction` is called with `sfactr` set to a value other than BF_ONE, or `dfactr` set to a value other than BF_ZERO, a more complex expression defines the frame buffer replacement algorithm.

```
Rdest = min (255, ((Rsource * sfactr) + (Rdest * dfactr)))
Gdest = min (255, ((Gsource * sfactr) + (Gdest * dfactr)))
Bdest = min (255, ((Bsource * sfactr) + (Bdest * dfactr)))
Adest = min (255, ((Asource * sfactr) + (Adest * dfactr)))
```

Each frame buffer color component is replaced by a weighted sum of the current value and the incoming pixel value. This sum is clamped to a maximum of 255. Blending factors `sfactr` and `dfactr` are chosen from the following list:

symbolic name	value in expression	notes
BF_ZERO	0.0	
BF_ONE	1.0	
BF_SA	Asource / 255	
BF_MSA	1.0 - (Asource / 255)	
BF_DA	Adestination / 255	requires alpha bitplanes
BF_MDA	1.0 - (Adestination / 255)	requires alpha bitplanes
BF_SC	RGBAsource / 255	dfactr only
BF_MSC	1.0 - (RGBAsource / 255)	dfactr only
BF_DC	RGBAdestination / 255	sfactr only
BF_MDC	1.0 -(RGBAdestination / 255)	sfactr only
BF_MIN_SA_MDA	min (BF_SA, BF_MDA)	requires alpha bitplanes, changes expression

Table 15–1. Blending Factors

All the blending factors are defined to have values between 0.0 and 1.0 inclusive. In most cases, this range is obtained by dividing a color component value, in the range 0 through 255, by 255. The blending arithmetic is done such that a blending factor with a value of 1.0 has no effect on the color component that it weights. Also, a blending factor of 0.0 forces its corresponding color component to 0. Because the weighting factors are all positive, and because each weighted sum is clamped to 255, colors blend toward white, rather than wrapping back to low-intensity values.

Blending factors BF_DA, BF_MDA, and BF_MIN_SA_MDA require alpha bitplanesfor correct operation. You can check to see if your machine has alpha bitplanes by calling:

```
getgdesc(GD_BITS_NORM_SNG_ALPHA)
```

and testing for a non-zero return value. Blending functions specified without using these three symbolic constants work correctly, regardless of the availability of alpha bitplanes.

Blending factors BF_SC, BF_MSC, BF_DC, and MF_MDC weight each color component by the corresponding weight component. For example, you can scale each framebuffer color component by the incoming color component with the blending function:

```
blendfunction (BF_DC,BF_ZERO)

Rdestination = min (255, (Rsource * (Rdestination / 255)))
Gdestination = min (255, (Gsource * (Gdestination / 255)))
Bdestination = min (255, (Bsource * (Bdestination / 255)))
Adestination = min (255, (Asource * (Adestination / 255)))
```

The special blending factor BF_MIN_SA_MDA is intended to support polygon antialiasing, as described in Section 15.3.3. It must be used only for `sfactr`, and only while `dfactr` is BF_ONE. In this case, the blending equations are:

```
blendfunction (BF_MIN_SA_MDA,BF_ONE)

Rdestination = min (255, ((Rsource * sfactr) + Rdestination)
Gdestination = min (255, ((Gsource * sfactr) + Gdestination)
Bdestination = min (255, ((Bsource * sfactr) + Bdestination)
sfactr = mind ((Asource/255), (1.0 - (Adestination/255)))
Adestination = sfactr + Adestination
```

This special blending function accumulates pixel contributions until the pixel is fully specified, then allows no further changes. Frame buffer alpha bitplanes, which must be present, store the accumulated contribution percentage, or "coverage".

Although many blending functions are supported, the function

```
blendfunction (BF_SA,BF_MSA)
```

stands out as the single most useful one. It weights incoming color components by the incoming alpha value, and frame buffer components by one minus the incoming alpha value. In other words, it blends between the incoming color and the frame buffer color, as a function of the incoming alpha. This function renders transparent objects by drawing them from farthest to nearest, specifying opacity as incoming alpha. Most of the antialiased primitives described in the next section also use this function.

The example below illustrates a different use of blending: image composition. Three images, represented by colored circles, are blended such that the first image is weighted by 0.5, the second by 0.35, and the third by 0.15. Blending function

```
blendfunction (BF_SA,BF_ONE)
```

is used, causing the circles to be added to each other, rather than blended in the sense of the paragraph above. Thus the order in which the circles are drawn makes no difference. Because the three weights add up to exactly 1.0, no clamping is done.

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

#define WINSIZE  400
#define RGB_BLACK0x000000
#define RGB_RED  0x0000ff
#define RGB_GREEN0x00ff00
#define RGB_BLUE 0xff0000

main()
{
    if (getgdesc(GD_BITS_NORM_SNG_RED) == 0 {
        fprintf(sterr, "Single buffered RGB not available\n");
        return 1;
    }
    if (getgdesc(GD_BLEND) == 0) {
        fprintf(sterr, "Blending not available\n");
        return 1;
    }
```

```
    prefsize(WINSIZE,WINSIZE);
    winopen("blendcircs");
    mmode(MVIEWING);
    RGBmode();
    gconfig();
    mmode(MVIEWING);
    ortho2(-1.0, 1.0, -1.0, 1.0);
    glcompat(GLC_OLDPOLYGON, 0);
    blendfunction(BF_SA, BF_ONE);
    cpack(RGB_BLACK);
    clear();
    cpack(0x80000000 | RGB_RED);   /* red with alpha = 128/255 */
    circf(0.25, 0.0, 0.7);
    sleep(2);
    cpack(0x4f000000 | RGB_GREEN); /* green with alpha =79/255 */
    circf(-0.25, 0.25, 0.7);
    sleep(2);
    cpack(0x30000000 | RGB_BLUE);  /* blue with alpha = 48/255 */
    circf(-0.25, -0.25, 0.7);
    sleep(10);
    gexit();
    return 0;
}
```

15.3 One-Pass Antialiasing—the Smooth Primitives

Aliasing artifacts are especially objectionable in image animations, because jaggies often introduce motion unrelated to the actual direction of motion of the primitives. The techniques described in this section improve the sampling quality of primitives without requiring that the primitives be drawn more than once. These techniques therefore perform well enough to animate complex scenes.

Modes are provided to support the drawing of antialiased points and lines in both color map and RGB modes. Because their interactions are more critical to the antialiasing quality, antialiased polygons are supported only in the more general RGB mode. If you are drawing an image composed entirely of points and/or lines, the routines in this section are always the right choice for antialiasing. If you include polygons in the image, you should consider both the techniques described in this section and the multipass accumulation technique described in Section 15.4.

15.3.1 High-Performance Antialiased Points—pntsmooth

Not all systems support `pntsmooth`. Refer to the manual page for details.

By default, IRIS-4D Series systems sample points by selecting and drawing the pixel nearest the exact projection of the mathematical point. You can improve the sampling quality of points, and therefore draw them antialiased, by setting two modes:

```
subpixel(TRUE);
pntsmooth(SMP_ON);
```

When you enable `subpixel` mode, you defeat the default behavior of rounding projected vertices to the nearest pixel center. Exact point position is made available to the sampling hardware. By enabling `pntsmooth` mode, you replace the default sampling of points with coverage sampling of a unit-diameter circle centered around the exact mathematical point position. All that remains is instructing the system on how to use the computed pixel coverage to blend between the background color and the point color at each pixel. This instruction differs based on whether the drawing is done in color map mode or in RGB mode.

When you enable `pntsmooth` while in color map mode, the antialiasing hardware uses computed pixel coverage to replace the 4 least significant bits of the point's color. Therefore, for color map antialiased points to blend correctly, you must initialize a 16-entry colormap block (whose lowest entry location is a multiple of 16) to a ramp between the background color (lowest index) and the point color (highest index). Before drawing points, clear the background to the same color used as background in the colormap ramp.

When you draw a point with a color index in the range of the specified ramp, pixels in the area of the exact mathematical point are written with color indices that select ramp values based on the fraction of the pixel that is obscured by the point's unit-diameter circle. Because the sampling hardware modifies only the 4 least significant bits of the point's color, you can initialize and use multiple color ramps, each with a different point color, in the same image. Note that all ramps must blend to the same background color, which must be the color of the background used for the image.

The following example illustrates the difference in image quality when you use `pntsmooth` and `subpixel` together to antialias color map points. The antialiased points and lines drawn by these example programs look better if you set gamma correction to 2.4, instead of the default value of 1.7 (type **`gamma 2.4`** in any *wsh* window on the screen).

```
/*
 *Drag a string of color map antialiased points with the cursor.
 *Disable antialiasing while the left mouse button is pressed.
 *Disable subpixel positioning while the middle mouse button is
 *pressed.
 */

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

#define WINSIZE     400
#define RAMPBASE    64      /* avoid the first 64 colors */
#define RAMPSIZE    16
#define RAMPSTEP    (255 / (RAMPSIZE-1))
#define MAXPOINT    25

Device devs[2] = {MOUSEX,MOUSEY};
```

```
main()
{
    short val, vals[2];
    long i, xorg, yorg;
    float vert[2], x, y, interp;

    if (getgdesc(GD_PNTSMOOTH_CMODE) == 0) {
       fprintf(stderr, "Color map mode point antialiasing not\
       available\n");
       return 1;
    }
    if (getgdesc(GD_BITS_NORM_DBL_CMODE) < 8) {
       fprintf(stderr, "Need 8 bitplanes in doublebuffer color\
       map mode\n");
       return 1;
    }
    prefsize(WINSIZE, WINSIZE);
    winopen("pntsmooth.index");
    mmode(MVIEWING);
    ortho2(-0.5, WINSIZE-0.5, -0.5, WINSIZE-0.5);
    doublebuffer();
    gconfig();
    qdevice(ESCKEY);
    qdevice(LEFTMOUSE);
    qdevice(MIDDLEMOUSE);
    getorigin(&xorg, &yorg);
    for (i = 0; i < RAMPSIZE; i++)
       mapcolor(i + RAMPBASE, i * RAMPSTEP, i * RAMPSTEP,
       i * RAMPSTEP);

    while (!(qtest() && qread(&val) == ESCKEY && val == 0)) {
       color(RAMPBASE);
       clear();
       getdev(2,devs,vals);
       x = vals[0] - xorg;
       y = vals[1] - yorg;
       pntsmooth(getbutton(LEFTMOUSE) ? SMP_OFF : SMP_ON);
       subpixel(getbutton(MIDDLEMOUSE) ? FALSE : TRUE);
       color(RAMPBASE+RAMPSIZE-1);
```

```
            bgnpoint();
                for (i=0; i<=MAXPOINT; i++) {
                    interp = (float)i / (float)MAXPOINT;
                    vert[0] = 100.0 * interp + x * (1.0 - interp);
                    vert[1] = 100.0 * interp + y * (1.0 - interp);
                    v2f(vert);
                }
            endpoint();
            swapbuffers();
        }
        gexit();
        return 0;
}
```

Notice how smoothly the antialiased points move as you move the cursor. Now defeat the antialiasing by pressing the left mouse button, and notice that the points move less smoothly, and that they do not line up nearly as well as the antialiased points. The image quality degrades in exactly the same way when you defeat subpixel positioning by pressing the middle mouse button.

The antialiased points look good when they are not drawn touching each other. However, when you move the cursor near the lower-left corner of the window, causing the points to bunch together, the image quality again degrades. This is because pixels that are obscured by more than one point take as their value the color computed for the last point drawn. There is no general solution to the problem of overlapping primitives while drawing in color map mode.

The problem of overlapping primitives is handled well when antialiasing in RGB mode. When you enable `pntsmooth` in RGB mode, the antialiasing hardware uses computed pixel coverage to scale the alpha value of the point's color. If the alpha value of the incoming point is 1.0, scaling it by the computed pixel coverage results in a pixel alpha value that is directly proportional to pixel coverage. For RGB antialiased points to draw correctly, set `blendfunction` to merge new pixel color components into the frame buffer using the incoming alpha value.

The following example illustrates RGB mode point antialiasing:

```
/*
 * Drag a string of RGB antialiased points with the cursor.
 * Change from a merge-blend to an accumulate-blend when the
 * left mouse button is depressed.
 * Use the "smoother" antialiasing sampling algorithm when the
 * middle mouse button is depressed.
 */

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

#define WINSIZE     400
#define MAXPOINT    25

Device devs[2] = {MOUSEX,MOUSEY};

main()
{
    short val, vals[2];
    long i, xorg, yorg;
    float vert[2], x, y, interp;

    if (getgdesc(GD_PNTSMOOTH_RGB) == 0) {
        fprintf(stderr, "RGB mode point antialiasing not \
        available\n");
    return 1;
    }
    prefsize(WINSIZE, WINSIZE);
    winopen("pntsmooth.rgb");
    mmode(MVIEWING);
    ortho2(-0.5, WINSIZE-0.5, -0.5, WINSIZE-0.5);
    doublebuffer();
    RGBmode();
    gconfig();
    qdevice(ESCKEY);
    qdevice(LEFTMOUSE);
    qdevice(MIDDLEMOUSE);
    getorigin(&xorg, &yorg);
    subpixel(TRUE);
```

```
    while (!(qtest() && qread(&val) == ESCKEY && val == 0)) {
        cpack(0);
        clear();
        getdev(2,devs,vals);
        x = vals[0] - xorg;
        y = vals[1] - yorg;
        cpack(0xffffffff);
        blendfunction(BF_SA,getbutton(LEFTMOUSE) ?
            BF_ONE : BF_MSA);
        pntsmooth(getbutton(MIDDLEMOUSE) ?
            (SMP_ON | SMP_SMOOTHER) : SMP_ON);
        bgnpoint();
            for (i=0; i<=MAXPOINT; i++) {
                interp = (float)i / (float)MAXPOINT;
                vert[0] = 100.0 * interp + x * (1.0 - interp);
                vert[1] = 100.0 * interp + y * (1.0 - interp);
                v2f(vert);
            }
        endpoint();
        swapbuffers();
    }
    gexit();
    return 0;
}
```

Unlike the color map antialiased points, the RGB antialiased points look good when they are bunched together. This is because RGB blending allows multiple points to contribute to a single pixel in a meaningful way. In this demonstration a blend function that interpolates between the incoming and frame buffer color components, based on the incoming alpha, is used by default.

```
blendfunction(BF_SA,BF_MSA)
```

Press the left mouse button to switch to a blend function that simply accumulates color, again scaled by incoming alpha:

```
blendfunction(BF_SA,BF_ONE)
```

The difference between `blendfunction(BF_SA,BF_MSA)` and `blendfunction(BF_SA,BF_ONE)` is more apparent when you draw lines (see Section 15.3.2). In this demonstration, note that bunched points are a little brighter when you select the accumulating blending function

You can switch from the standard antialiasing sampling algorithm to a "smoother" algorithm by pressing the middle mouse button. Not all systems support the higher-quality point sampling algorithm. Refer to the manal page for details. This algorithm modifies more pixels per antialiased point than does the standard antialiasing algorithm. As a result, it produces slightly higher-quality antialiased points, at the price of slightly reduced performance. Set the "smoother" algorithm by calling

```
pntsmooth(SMP_ON | SMP_SMOOTHER);
```

Because RGB mode antialiased points are blended into the frame buffer, they can be drawn in any color and over any background. Unless you want to draw transparent, antialiased points, however, be sure to specify alpha as 1.0 when drawing antialiased RGB points.

15.3.2 High-Performance Antialiased Lines—linesmooth

Not all systems support `linesmooth`. Refer to the manual page for details.

By default, IRIS-4D Series systems sample lines by selecting and drawing the pixels nearest the projection of the mathematical line. You can improve the sampling quality of lines, and therefore draw them antialiased, by setting two modes:

```
subpixel(TRUE);
linesmooth(SML_ON);
```

When you enable `subpixel` mode, you defeat the default behavior of rounding projected vertices to the nearest pixel center . Exact line endpoint position is made available to the sampling hardware. By enabling `linesmooth` mode, you replace the default sampling of lines with coverage sampling of a unit-width rectangle centered around the exact mathematical line. All that remains is instructing the system on how to use the computed pixel coverage to blend between the background color and the line color at each pixel. This instruction differs based on whether the drawing is done in color map mode or in RGB mode.

When you enable `linesmooth` while in color map mode, the antialiasing hardware uses computed pixel coverage to replace the 4 least significant bits of the line's color. Therefore, for color map antialiased lines to appear correct, you must initialize a 16-entry colormap block (whose lowest entry location is a multiple of 16) to a ramp between the background color (lowest index) and the line color (highest index). Before drawing lines, clear the background to the same color used as background in the color map ramp.

When you draw a line with a color index in the range of the specified ramp, pixels in the area of the exact mathematical line are written with color indices that select ramp values based on the fraction of the pixel that is obscured by the line's unit-width rectangle. Because the sampling hardware modifies only the 4 least significant bits of the line's color, you can initialize and use multiple color ramps, each with a different line color, in the same image. Note that all ramps must blend to the same background color, which must be the color of the background used for the image.

The following example illustrates the difference in image quality when you use `linesmooth` and `subpixel` together to antialias color map lines. The program draws a single straight line, made up of several individual line segments.

```
/*
 * Drag a string of color map antialiased line segments with
 * the cursor. Disable antialiasing while the left mouse
 * button is pressed. Disable subpixel positioning while
 * the middle mouse button is pressed.
 */

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

#define WINSIZE     400
#define RAMPBASE    64      /* avoid the first 64 colors */
#define RAMPSIZE    16
#define RAMPSTEP    (255 / (RAMPSIZE-1))
#define MAXVERTEX    10

Device devs[2] = {MOUSEX,MOUSEY};
```

```
main()
{
    short val, vals[2];
    long i, xorg, yorg;
    float vert[2], x, y, interp;

    if (getgdesc(GD_LINESMOOTH_CMODE) == 0) {
       fprintf(stderr, "Color map mode line antialiasing not\
       available\n");
       return 1;
    }
    if (getgdesc(GD_BITS_NORM_DBL_CMODE) < 8) {
       fprintf(stderr, "Need 8 bitplanes in doublebuffer color\
       map mode\n");
       return 1;
    }
    prefsize(WINSIZE, WINSIZE);
    winopen("linesmooth.index");
    mmode(MVIEWING);
    ortho2(-0.5, WINSIZE-0.5, -0.5, WINSIZE-0.5);
    doublebuffer();
    gconfig();
    qdevice(ESCKEY);
    qdevice(LEFTMOUSE);
    qdevice(MIDDLEMOUSE);
    getorigin(&xorg, &yorg);
    for (i = 0; i < RAMPSIZE; i++)
       mapcolor(i + RAMPBASE, i * RAMPSTEP, i * RAMPSTEP,
       i * RAMPSTEP);

    while (!(qtest() && qread(&val) == ESCKEY && val == 0)) {
       color(RAMPBASE);
       clear();
       getdev(2,devs,vals);
       x = vals[0] - xorg;
       y = vals[1] - yorg;
       linesmooth(getbutton(LEFTMOUSE) ? SML_OFF : SML_ON);
       subpixel(getbutton(MIDDLEMOUSE) ? FALSE : TRUE);
       color(RAMPBASE+RAMPSIZE-1);
```

```
        bgnline();
            for (i=0; i<=MAXVERTEX; i++) {
                interp = (float)i / (float)MAXVERTEX;
                vert[0] = 100.0 * interp + x * (1.0 - interp);
                vert[1] = 100.0 * interp + y * (1.0 - interp);
                v2f(vert);
            }
        endline();
        swapbuffers();
    }
    gexit();
    return 0;
}
```

Notice how smooth the edges of the antialiased lines are, and how smoothly they move as you move the cursor. Now defeat the antialiasing by pressing the left mouse button, and notice that the lines become jagged. When you defeat subpixel positioning by pressing the middle mouse button, the individual line segments that make up the long line remain antialiased, but they no longer combine to form a single straight line. This is because the endpoints of the segments have been coerced to the nearest pixel centers, which are rarely on the exact mathematical line. Thus, you can antialias lines, unlike points, while subpixel mode is `FALSE`. However, the image quality is still greatly enhanced when you enable subpixel positioning of vertices.

Like color map antialiased points, color map antialiased lines interact poorly when they intersect on the screen. The problem of overlapping primitives is handled well when antialiasing in RGB mode. When you enable `linesmooth` in RGB mode, the antialiasing hardware uses computed pixel coverage to scale the alpha value of the line's color. If the alpha value of the incoming line is 1.0, scaling it by the computed pixel coverage results in a pixel alpha value that is directly proportional to pixel coverage. For RGB antialiased lines to draw correctly, set `blendfunction` to merge new pixel color components into the frame buffer using the incoming alpha value.

The following example illustrates RGB mode line antialiasing:

```
/*
 * Rotate a pinwheel of antialiased lines drawn in RGB mode.
 * Change to the "smoother" sampling function when the left
 * mouse button is pressed. Change to the "end-corrected"
 * sampling function when the middle mouse button is
 * pressed. Change to a "color index like" blend function
 * when the i-key is pressed. Change from merge-blend to
 * accumulate-blend when the a-key is pressed.
 * Disable subpixel positioning when the s-key is pressed.
 */

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

#define WINSIZE    400
#define MAXLINE    48
#define ROTANGLE   (360.0 / MAXLINE)

float vert0[2] = {0.0,0.0};
float vert1[2] = {0.8,0.0};

main()
{
    int i;
    int smoothmode;
    short val;

    if (getgdesc(GD_LINESMOOTH_RGB) == 0) {
       fprintf(stderr, "RGB mode line antialiasing not\
       available\n");
       return 1;
    }
    prefsize(WINSIZE, WINSIZE);
    winopen("linesmooth.rgb");
    mmode(MVIEWING);
    ortho2(-1.0,1.0,-1.0,1.0);
    doublebuffer();
    RGBmode();
    gconfig();
    qdevice(ESCKEY);
```

```
    qdevice(LEFTMOUSE);
    qdevice(MIDDLEMOUSE);

    while (!(qtest() && qread(&val) == ESCKEY && val == 0)) {
        cpack(0);
        clear();
        cpack(0xffffffff);
        smoothmode = SML_ON;
        if (getbutton(LEFTMOUSE))
            smoothmode |= SML_SMOOTHER;
        if (getbutton(MIDDLEMOUSE))
            smoothmode |= SML_END_CORRECT;
        linesmooth(smoothmode);
        if (getbutton(IKEY))
            blendfunction(BF_SA,BF_ZERO);
        else if (getbutton(AKEY))
            blendfunction(BF_SA,BF_ONE);
        else
            blendfunction(BF_SA,BF_MSA);
        subpixel(getbutton(SKEY) ? FALSE : TRUE);
        pushmatrix();
        rot(getvaluator(MOUSEX) / 25.0,'z');
        for (i=0; i<MAXLINE; i++) {
            bgnline();
                v2f(vert0);
                v2f(vert1);
            endline();
            rot(ROTANGLE,'z');
        }
        popmatrix();
        swapbuffers();
    }
    gexit();
    return 0;
}
```

Notice that the RGB mode antialiased lines look good where they intersect at the center of the pinwheel. This is because RGB blending allows multiple lines to contribute to a single pixel in a meaningful way. In this demonstration a blend function that interpolates between the incoming and frame buffer color components, based on the incoming alpha, is used by default:

```
blendfunction(BF_SA,BF_MSA)
```

You can switch to a blend function that simply accumulates color, again scaled by incoming alpha, by pressing the <A> key:

```
blendfunction(BF_SA,BF_ONE)
```

This blending function makes the slight noise at the center of the pinwheel disappear, because these pixels all accumulate and clamp at full brightness. This technique works well with white lines on a black background, but does not do well in other situations.

You can simulate the appearance of color map mode lines by pressing the <i> key, which forces a blending function that overwrites pixels:

```
blendfunction(BF_SA,BF_ZERO);
```

When you defeat subpixel positioning of line endpoints by pressing the <s> key, the pinwheel ceases to behave like a rigid object, and instead appears to wiggle and twist as it is rotated.

You can switch from the standard antialiasing sampling algorithm to a "smoother" algorithm by pressing the left mouse button. Not all systems support the higher-quality line sampling algorithm. Refer to the man page for details. This algorithm modifies more pixels per unit line length than does the standard antialiasing algorithm. As a result, it produces slightly higher-quality antialiased lines, at the price of slightly reduced performance. Set the "smoother" algorithm by calling `linesmooth (SMP_ON | SMP_SMOOTHER)`.

Notice that when it is selected, lines at all angles appear to have the same width, and the "cloverleaf" pattern at the center of the pinwheel disappears. When you rotate the pinwheel with the left mouse button pressed, the only image artifact that remains is the sudden changing of line length observed at the ends of the lines. Press the middle mouse button to select a sampling algorithm that correctly samples line length as well as line cross-section. Invoke this "end-corrected" algorithm by calling `linesmooth(SMP_ON | SMP_END_CORRECT)`.

When you select both "smoother" and "end-correct", the rotating pinwheel appears absolutely rigid, with even width lines and no jagged edges.

Caution: Because RGB antialiased lines are blended, they interact well at intersections. However, when two RGB antialiased lines are drawn between the same vertices, the line quality is reduced noticeably. When the polygons in a standard geometric model are drawn as lines, either explicitly or using `polymode`, lines at the edges of adjacent polygons are drawn twice, and therefore do not antialiased well in RGB mode. For best results, modify the database traversal so that edges of adjacent polygons are drawn only once.

Because RGB antialiased lines are blended into the frame buffer, they can be drawn in any color over any background. Unless you want to draw transparent, anti-aliased lines, however, be sure to specify alpha as 1.0 when drawing antialiased RGB lines.

15.3.3 High-Performance Antialiased Polygons—polysmooth

Not all systems support `polysmooth`. Refer to the manual page for details. By default, IRIS-4D Series systems sample polygons by selecting and drawing the pixels whose exact center points are within the boundary described by the projection of the mathematical polygon edges. You can improve the sampling quality of polygons, and therefore draw them antialiased, by setting two modes:

```
subpixel(TRUE);
polysmooth(PYSM_ON);
```

When you enable `subpixel` mode, you defeat the default behavior of rounding projected vertices to the nearest pixel center. Exact polygon vertex positions are made available to the sampling hardware. By enabling polysmooth mode, you replace the default sampling of polygons with coverage sampling—the fraction of each pixel covered by the polygon is computed. All that remains is instructing the system how to use the computed pixel coverage to blend between the background color and the polygon color at each pixel. Because this blending operation is more critical for polygon antialiasing than it is for point or line antialiasing, polygon antialiasing is supported only in RGB mode, not in color map mode.

The following program draws a single antialiased triangle:

```
/*
 * Rotate a single antialiased triangle drawn in RGB mode.
 * Disable antialiasing when the left mouse button is pressed.
 * Disable subpixel positioning when the middle mouse button
 * is pressed.
 */

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

#define WINSIZE 400

float vert0[2] = {0.0,0.0};
float vert1[2] = {0.8,0.0};
float vert2[2] = {0.4,0.4};
```

```
main()
{
    short val;

    if (getgdesc(GD_POLYSMOOTH) == 0) {
       fprintf(stderr, "polygon antialiasing not available\n");
       return 1;
    }
    prefsize(WINSIZE, WINSIZE);
    winopen("polysmooth.rgb");
    mmode(MVIEWING);
    ortho2(-1.0,1.0,-1.0,1.0);
    doublebuffer();
    RGBmode();
    gconfig();
    qdevice(ESCKEY);
    qdevice(LEFTMOUSE);
    qdevice(MIDDLEMOUSE);
    blendfunction(BF_SA,BF_MSA);

    while (!(qtest() && qread(&val) == ESCKEY && val == 0)) {
       cpack(0);
       clear();
       cpack(0xffffffff);
       polysmooth(getbutton(LEFTMOUSE) ? PYSM_OFF : PYSM_ON);
       subpixel(getbutton(MIDDLEMOUSE) ? FALSE : TRUE);
       pushmatrix();
       rot(getvaluator(MOUSEX) / 25.0,'z');
       rot(getvaluator(MOUSEY) / 10.0,'x');
       bgnpolygon();
          v2f(vert0);
          v2f(vert1);
          v2f(vert2);
       endpolygon();
       popmatrix();
       swapbuffers();
    }
    gexit();
    return 0;
}
```

Move the cursor left and right to rotate the triangle, and note the smoothness of its edges. When you move the cursor toward the top of the screen, the triangle rotates away from you until it becomes perpendicular to your viewing direction. Note that when it is perpendicular, it disappears completely. This is because the projection of a triangle on edge covers no area on the screen, and therefore all pixel coverages are zero.

When you press the left mouse button, the triangle is drawn aliased. When you press the middle mouse button , the triangle vertices are no longer subpixel positioned. Notice that the edges remain smooth, but that the triangle motion is no longer smooth, and the triangle no longer appears rigid.

This simple example of a single antialiased triangle drawn on a black background works correctly with the standard blending function:

```
blendfunction(BF_SA,BF_MSA)
```

However, when multiple antialiased triangles are drawn with adjacent edges, the standard blending function no longer produces good results. The following program draws a bowtie-shaped object, constructed of four triangles in a planar mesh:

```
/*
 * Rotate a patch of antialiased triangles drawn in RGB mode.
 * Disable special polygon-blend when the left mouse button
 * is pressed. Disable subpixel positioning when the middle
 * mouse button is pressed.
 */

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

#define WINSIZE 400

float vert0[2] = {0.0,0.0};
float vert1[2] = {0.0,0.4};
float vert2[2] = {0.4,0.1};
float vert3[2] = {0.4,0.3};
float vert4[2] = {0.8,0.0};
float vert5[2] = {0.8,0.4};
```

```
main()
{
    short val;

    if (getgdesc(GD_POLYSMOOTH) == 0) {
       fprintf(stderr, "polygon antialiasing not available\n");
       return 1;
    }
    prefsize(WINSIZE, WINSIZE);
    winopen("polysmooth2.rgb");
    mmode(MVIEWING);
    ortho(-1.0,1.0,-1.0,1.0,-1.0,1.0);
    doublebuffer();
    RGBmode();
    gconfig();
    qdevice(ESCKEY);
    qdevice(LEFTMOUSE);
    qdevice(MIDDLEMOUSE);
    polysmooth(PYSM_ON);
    shademodel(FLAT);

    while (!(qtest() && qread(&val) == ESCKEY && val == 0)) {
       cpack(0);
       clear();
       cpack(0xffffffff);
       if (getbutton(LEFTMOUSE))
           blendfunction(BF_SA,BF_MSA);
       else
           blendfunction(BF_MIN_SA_MDA,BF_ONE);
       subpixel(getbutton(MIDDLEMOUSE) ? FALSE : TRUE);
       pushmatrix();
       rot(getvaluator(MOUSEX) / 25.0,'z');
       rot(getvaluator(MOUSEY) / 10.0,'x');
       bgntmesh();
           v2f(vert0);
           v2f(vert1);
           v2f(vert2);
           v2f(vert3);
           v2f(vert4);
           v2f(vert5);
       endtmesh();
       popmatrix();
       swapbuffers();
    }
```

```
    gexit();
    return 0;
}
```

Notice that the internal edges of the four triangles that make up the bowtie are visible. Press the left mouse button, enabling the special polygon blending function, and note that these internal edges disappear. They are visible when you use the standard blending function because the standard blending function operates with uncorrelated coverages, such as those generated by antialiased points and lines. Two adjacent polygons generate pixel coverages that are highly correlated—they always sum to 100% for pixels covered by the shared edge—and are therefore inappropriate for the standard blending function. Consider, for example, a pixel that is covered 60% by the first polygon that intersects it, and 40% by a second polygon adjacent to the first. Assuming white polygons and a black background, the first polygon raises the pixel intensity to 0.6, which is the correct value. However, the second polygon raises the pixel intensity to only 0.76, rather than to 1.0 as is desired. This is because the standard blending function assumes that the 60% and 40% coverages are uncorrelated, so 60% of the additional 40% is assumed to have been covered by the original 60%. Thus in uncorrelated coverage arithmetic, 60% plus 40% equals 76%, not 100%.

The special blending function `blendfunction(BF_MIN_SA_MDA, BF_ONE)` works with correlated coverages, the kind generated by antialiased polygon images. As the example code illustrated, the correlated blend does a good job with polygonal data. It is, however, much more difficult to use correlated blending than uncorrelated blending. The requirements for its use are:

1. You must have alpha bitplanes.

2. You must draw polygons in order from the nearest to the farthest.

3. You must not draw backfacing polygons (use `backface`).

4. The background color bitplanes, including the alpha bitplanes, must be cleared to zero before drawing starts.

5. If the background is any color other than black, it must be filled as a polygon (i.e. not with a `clear` command) after all polygons are drawn.

6. You must draw all primitives (points, lines, and polygons) using the correlated blending function.

The correlated blending function works by accumulating pixel coverage in the frame buffer alpha bitplanes. The coverage granted each pixel write is limited by the total remaining at that pixel. When no coverage is left, additional writes to that pixel are ignored.

Because polygons must be drawn in depth-sorted order, you cannot use the z-buffer to eliminate hidden surfaces. Thus, polygon antialiasing, unlike point and line antialiasing, requires significant changes to the way the object data are traversed. It is therefore more difficult to use than are point and line antialiasing. If performance is not an absolute requirement, the accumulation buffer technique described in Section 15.4 is a better choice for polygon antialiasing.

The following example demonstrates correct polygon antialiasing of two cubes against a non-black background:

```
/*
 * Rotate two antialiased cubes in RGB mode.
 * Disable antialiasing by depressing the left mouse button.
 */

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

#define WINSIZE 400
#define SIZE  (0.2)
#define OFFSET   (0.5)
#define CUBE0 OFFSET
#define CUBE1 (-OFFSET)

float vert0[4] = {-SIZE,-SIZE, SIZE};
float vert1[4] = { SIZE,-SIZE, SIZE};
float vert2[4] = {-SIZE, SIZE, SIZE};
float vert3[4] = { SIZE, SIZE, SIZE};
float vert4[4] = {-SIZE, SIZE,-SIZE};
float vert5[4] = { SIZE, SIZE,-SIZE};
float vert6[4] = {-SIZE,-SIZE,-SIZE};
float vert7[4] = { SIZE,-SIZE,-SIZE};
```

```
float cvert0[2] = {-1.0,-1.0};
float cvert1[2] = { 1.0,-1.0};
float cvert2[2] = { 1.0, 1.0};
float cvert3[2] = {-1.0, 1.0};

main()
{
    short val;
    float xang;

    if (getgdesc(GD_POLYSMOOTH) == 0) {
       fprintf(stderr, "polygon antialiasing not available\n");
       return 1;
    }
    prefsize(WINSIZE, WINSIZE);
    winopen("polysmooth3.rgb");
    mmode(MVIEWING);
    ortho(-1.0,1.0,-1.0,1.0,-1.0,1.0);
    doublebuffer();
    RGBmode();
    gconfig();
    qdevice(ESCKEY);
    qdevice(LEFTMOUSE);
    blendfunction(BF_MIN_SA_MDA,BF_ONE);
    subpixel(TRUE);
    backface(TRUE);
    shademodel(FLAT);

    while (!(qtest() && qread(&val) == ESCKEY && val == 0)) {
       cpack(0);
       clear();
       polysmooth(getbutton(LEFTMOUSE) ? PYSM_OFF : PYSM_ON);
       pushmatrix();
       xang = getvaluator(MOUSEY) / 5.0;
       rot(xang,'x');
       rot(getvaluator(MOUSEX) / 5.0,'z');
       if (xang < 90.0) {
           drawcube(CUBE0);
           drawcube(CUBE1);
       } else {
           drawcube(CUBE1);
           drawcube(CUBE0);
       }
       popmatrix();
```

```
        drawbackground();
        swapbuffers();
    }
    gexit();
    return 0;
}

drawcube(offset)
float offset;
{
    pushmatrix();
    translate(0.0,0.0,offset);
    bgntmesh();
    v3f(vert0);
    v3f(vert1);
    cpack(0xff0000ff);
    v3f(vert2);
    v3f(vert3);
    cpack(0xff00ff00);
    v3f(vert4);
    v3f(vert5);
    cpack(0xffff0000);
    v3f(vert6);
    v3f(vert7);
    cpack(0xff00ffff);
    v3f(vert0);
    v3f(vert1);
    endtmesh();
    bgntmesh();
    v3f(vert0);
    v3f(vert2);
    cpack(0xffff00ff);
    v3f(vert6);
    v3f(vert4);
    endtmesh();
    bgntmesh();
    v3f(vert1);
    v3f(vert7);
    cpack(0xffffff00);
    v3f(vert3);
    v3f(vert5);
    endtmesh();
    popmatrix();
}
```

```
drawbackground() {
    cpack(0xffffffff);
    bgnpolygon();
    v2f(cvert0);
    v2f(cvert1);
    v2f(cvert2);
    v2f(cvert3);
    endpolygon();
}
```

Note that the nearer cube is drawn first, that cube faces are not sorted because back face elimination handles the sorting of convex solids, and that the background is drawn last as a single polygon.

When you press the left mouse button, antialiasing is disabled, but the correlated blend function remains enabled. Otherwise, the drawing order of the primitives would have to be changed.

15.4 Multipass Antialiasing with the Accumulation Buffer

Section 15.3 describes techniques for computing pixel area coverage for various primitives, and using this coverage information to blend pixels into the framebuffer. This section describes an alternative approach to antialiasing. Called *accumulation*, it provides an elegant solution to the aliasing problem for points, lines, and polygons, and also has application in other advanced rendering techniques.

Accumulation is somewhat like blending, in that multiple images are composited to produce the final image. It differs from blending, however, in that its operation is completely separated from the rendering of a single frame. The accumulation buffer is an extended range bitplane bank in the normal frame buffer. You do not draw images into it; rather, images drawn in the front or back buffer of the normal frame buffer are added to the contents of the accumulation buffer after they are completely rendered.

acsize

Before you can use the accumulation buffer, you must allocate bitplanes for it. `acsize` specifies the number of bitplanes to be allocated for each color component in the accumulation buffer. Currently, 0 and 16 are the only sizes `acsize` accepts. A 16-bit accumulation buffer actually allocates 64 bitplanes, 16 each for red, green, blue, and alpha. Color components are signed values, so the range for each component is -32768 through 32767. You must call `gconfig` after `acsize` to activate the new specification.

After bitplanes have been allocated for the accumulation buffer, you can use the `acbuf` command to add the contents of the front or back bitplanes of the normal frame buffer to the accumulation buffer, and to return the accumulation buffer contents to either the front or back bitplanes. Call `acbuf` only while the normal frame buffer is in RGB mode.

```
void acsize(planes)
long planes;
```

acbuf

`acbuf` operates on the accumulation buffer, which must already have been allocated using `acsize` and `gconfig`. When `op` is `AC_CLEAR`, each component of the accumulation buffer is set to `value`. When `op` is `AC_ACCUMULATE`, pixels are taken from the current `readsource` (front, back, or z-buffer). Pixel components red, green, blue, and alpha are each scaled by <value>, which must be in the range -256.0 <= `value` <= 256.0, and added to the current contents of the accumulation buffer.

Finally, when `op` is `AC_RETURN`, pixels are taken from the accumulation buffer. Each pixel component is scaled by `value`, which must be in the range 0.0 through 1.0, clamped to the integer range 0 through 255, and returned to the currently active drawing buffer (as specified by the `frontbuffer`, `backbuffer`, and `zdraw` commands). All special pixel operations—including z-buffer, blending function, logical operation, stenciling, and texture mapping—are ignored during this transfer.

```
void acbuf(op,value)
long op;
float value;
```

(These commands implement several other accumulation buffer operations. See the manual page for details on these operations.)

Accumulation buffer pixels map one-to-one with pixels in the window. All accumulation buffer operations affect the pixels within the viewport, limited by the screen mask and by the edges of the window itself. Like front, back, and z-buffer pixels, accumulation buffer pixels corresponding to window pixels that are obscured by another window, or are not on the screen, are undefined.

You can use the accumulation buffer to average many renderings of the same scene into one final image. By jittering the viewing frustum slightly for each image, you can produce a single antialiased image as the result of many averaged images. For this to work, you must

1. Completely render the image for each pass, including using the z-buffer to eliminate hidden surfaces, if appropriate.

2. Enable subpixel positioning of all primitives used (see `subpixel`).

3. Slightly perturb the viewing frustum before rendering each image. By slightly perturbing the projection transformation before rendering each image, you can effectively move the sample position in each pixel away from the pixel center. This is particularly easy to implement when you use an orthographic projection.

The following example draws an antialiased circle using a 2-D orthographic projection:

```
/*
 * Draw an antialiased circle using the accumulation buffer.
 * Disable antialiasing when the left mouse button is pressed.
 * Disable subpixel positioning when the middle mouse button
 * is pressed.
 */

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

#define WINSIZE 100
#define SAMPLES 3
#define DELTA (2.0 / (WINSIZE * SAMPLES))

main()
{
    long x, y;
    short val;

    if (getgdesc(GD_BITS_ACBUF) == 0) {
       fprintf(stderr, "accumulation buffer not available\n");
       return 1;
    }
```

```
    prefsize(WINSIZE, WINSIZE);
    winopen("acbuf.rgb");
    mmode(MVIEWING);
    glcompat(GLC_OLDPOLYGON,0);    /* point sample the circle */
    doublebuffer();
    RGBmode();
    acsize(16);
    gconfig();
    qdevice(ESCKEY);
    qdevice(LEFTMOUSE);
    qdevice(MIDDLEMOUSE);

    while (!(qtest() && qread(&val) == ESCKEY && val == 0)) {
       subpixel(getbutton(MIDDLEMOUSE) ? FALSE : TRUE);
       if (getbutton(LEFTMOUSE)) {
           drawcirc(0.0,0.0);
       } else {
           acbuf(AC_CLEAR,0.0);
           for (x=0; x < SAMPLES; x++) {
              for (y=0; y < SAMPLES; y++) {
                  drawcirc((x-(SAMPLES/2))*DELTA,\
                  (y-(SAMPLES/2))*DELTA);
                  acbuf(AC_ACCUMULATE,1.0);
              }
           }
           acbuf(AC_RETURN,1.0/(SAMPLES*SAMPLES));
       }
       swapbuffers();
    }
    gexit();
    return 0;
}

drawcirc(xdelta,ydelta)
float xdelta,ydelta;
{
    ortho2(-1.0 + xdelta, 1.0 + xdelta, -1.0 + ydelta,\
    1.0 + ydelta);
    cpack(0);
    clear();
    cpack(0xffffffff);
    circf(0.0,0.0,0.8);
}
```

The circle is drawn nine times on a regular three-by-three grid. After the ninth accumulation, the resulting image is returned to the back buffer, and the buffers are swapped, making the antialiased circle visible. Note that the edges of the circle are smooth, and that when you press the left mouse button, the edges become jagged (they are now aliased). Also note the reduction in image quality when you defeat subpixel positioning by pressing the middle mouse button.

Some other points about this example:

1. The orthographic projection is perturbed by multiples of DELTA, a constant that is a function of the ratio of units in orthographic coordinates to window coordinates, and of the resolution of the subsampling. Note that you cannot use the viewport to jitter the sample points, both because the viewport is specified with integer coordinates, and because pixels near the viewport boundary would sample incorrectly.

2. Old polygon mode is defeated. Otherwise the circle would be drawn with the old fill style, rather than the new point-sampled style. Point sampling and subpixel positioning are both required to use the accumulation buffer accurately.

3. Each drawing pass clears the back buffer to black, then draws the circle. In general, all drawing operations (such as clearing and using the z-buffer) must be duplicated for each pass.

Of course, you can perform accumulation buffer antialiasing with perspective projections as well as orthographic projections. The following subroutines do all the arithmetic required to implement pixel jitter using the perspective and window projection calls:

```
#include <math.h>

void subpixwindow(left,right,bottom,top,near,far,pixdx,pixdy)
float left,right,bottom,top,near,far,pixdx,pixdy;

{
    short vleft,vright,vbottom,vtop;
    float xwsize,ywsize,dx,dy;
    int xpixels,ypixels;

    getviewport(&vleft,&vright,&vbottom,&vtop);
    xpixels = vright - vleft + 1;
    ypixels = vtop - vbottom + 1;
    xwsize = right - left;
    ywsize = top - bottom;
    dx = -pixdx * xwsize / xpixels;
    dy = -pixdy * ywsize / ypixels;
    window(left+dx,right+dx,bottom+dy,top+dy,near,far);
}

void subpixperspective(fovy,aspect,near,far,pixdx,pixdy)
Angle fovy;
float aspect, near, far, pixdx, pixdy;

{
    float fov2,left,right,bottom,top;
    fov2 = ((fovy*M_PI) / 1800) / 2.0;
    top = near / (fcos(fov2) / fsin(fov2));
    bottom = -top;
    right = top * aspect;
    left = -right;
    subpixwindow(left,right,bottom,top,near,far,pixdx,pixdy);
}
```

In many applications, you can condition use of the accumulation buffer on user input. For example, when mouse position determines view angle, you can accumulate and display a progressively higher-quality antialiased image while the mouse is stationary. At any time during an antialiasing accumulation, the contents of the accumulation buffer represent a better image than the aliased image. You might choose a sample pattern that optimizes the intermediate results, then display each intermediate result, rather than waiting for the accumulation to be complete.

The antialiasing example implements a box filter—samples are evenly distributed inside a square pixel, and each sample has the same effect on the resulting image. Antialiasing filter quality improves when the samples are distributed in a circular pattern that is larger than a pixel, perhaps with a diameter of 0.75 pixels or so. You can further improve the filter quality by shaping it as a symmetric Gaussian function, either by changing the density of sample locations within the circle, or by keeping the sample density constant and assigning different weights to the samples. The weight of a sample is specified by `<value>` when you call `acbuf(AC_ACCUMULATE, value)`. A circularly symmetric Gaussian filter function yields smoother edges than does a unit-size box filter.

Regardless of the filter function, fewer samples are required to achieve a given antialiasing quality level when the samples are distributed in a random fashion, rather than in regular rows and columns.

The accumulation buffer has many rendering applications other than antialiasing. For example, to limit depth of field, you can average images projected from slightly different viewpoints and directions. To produce motion blur, you can average images with moving objects rendered in different locations along their trajectories. To implement a filter kernel, you can convolve images with `rectcopy()` and the accumulation buffer. Because the accumulation buffer operates on signed color components, and clamps these components to the display range of 0 through 255 when they are returned to the display buffer, you can implement filters with negative components in their kernels.

Additional details of the theory and use of the accumulation buffer, as well as example images, are available in "The Accumulation Buffer: Hardware Support for High-Quality Rendering", in *SIGGRAPH '90 Conference Proceedings*, Volume 24, Number 3 (August 1990).

16. Graphical Objects

It is sometimes convenient to group together a sequence of drawing routines and give it an identifier. The entire sequence can then be repeated with a single reference to the identifier rather than by repeating all the drawing routines. In the Graphics Library, such sequences are called *graphical objects*; in other systems they are sometimes known as display lists. A graphical object is a list of graphics primitives (drawing routines) to display. For example, a drawing of an automobile can be viewed as a compilation of smaller drawings of each of its parts: windows, doors, wheels, etc. Each part (for example, a wheel) might be a graphical object—a series of `point`, `line`, and `polygon` routines.

To make the automobile a graphical object, you would first create objects that draw its parts—a wheel object, a door object, a body object, and so on. The automobile object would be a series of calls to the part objects, which together with appropriate rotation, translation, and scale routines, would put all the parts in their correct places.

16.1 Defining an Object

To create a graphical object, you call `makeobj`, call the same drawing routines you would normally call to draw the object, and then call `closeobj`. Between the `makeobj` and `closeobj` calls, drawing routines do not result in immediate drawing on the screen; rather, they are compiled into the object that is being created.

Thus, a graphical object is a list of primitive drawing routines to be executed. Drawing the graphical object consists of executing each routine in the listed order. There is no flow control, such as looping, iteration, or condition tests (except for `bbox2`,; see Section 16.2).

Note: Although the fast drawing routines (such as `bgn/endpoint`, `bgn/endline`, `n3f`, `c3f`, etc.) can now be included within graphical objects, not all Graphics Library routines can be included within a graphical object. A general rule is to include drawing routines and not to include routines that return values. If you have a question about a particular routine, check its *man* page in the appropriate Reference Guide.

makeobj

`makeobj` creates a graphical object. It takes one argument, a 31-bit integer that is associated with the object. If *obj* is the number of an existing object, the contents of that object are deleted.

When `makeobj` executes, the object number is entered into a symbol table and an empty graphical object is created. Subsequent graphics routines are compiled into the graphical object instead of being executed immediately.

```
void makeobj (obj)
Object obj;
```

closeobj

`closeobj` terminates the object definition and closes the open object. All the routines in the graphical object between `makeobj` and `closeobj` are part of the object definition.

```
void closeobj()
```

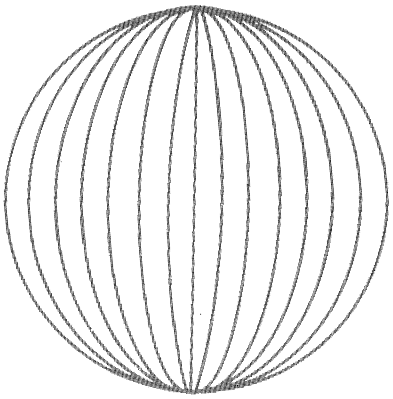
Figure 16-1 shows an object definition of a simple shape named sphere, and the figure it draws when called.

If you specify a numeric identifier that is already in use, the system replaces the existing object definition with the new one. To ensure that your object's numeric identifier is unique, use `isobj` and `genobj`.

isobj

`isobj` tests whether there is an existing object with a given numeric identifier. Its argument *obj* specifies the desired numeric identifier. `isobj` returns TRUE if an object exists with the specified numeric identifier and FALSE if none exists.

```
long isobj(obj)
Object obj;
```



```
makeobj(sphere=genobj ());
for (phi=0; phi<PI; phi+=PI/16) {
    bgnclosedline();
    for(theta=0; theta<2*PI; theta+=PI/18) {
        vert[0] = sin(theta) * cos(phi);
        vert[1] = sin(theta) * sin(phi);
        vert[2] = cos(theta);
        v3f(vert);
    }
    endclosedline();
}
closeobj();
```

The sphere above is defined as a graphical object. `makeobj` creates a new object containing Graphics Library routines between `makeobj` and `closeobj`.

Figure 16-1. Object Definition for a Simple Shape (Sphere)

genobj

`genobj` generates a unique numeric identifier. It does not generate current numeric identifiers. `genobj` is useful in naming objects when it is impossible to anticipate what the current numeric identifier will be when the routine is called.

```
Object genobj()
```

delobj

`delobj` deletes an object. It frees all memory storage associated with the object. The numeric identifier is undefined until it is reused to create a new object. The system ignores calls to deleted or undefined objects.

```
delobj(obj)
Object obj;
```

16.2 Using Objects

Once you create an object, you can use `callobj` to draw it. You can enable dynamic bounding box pruning and minimum feature size culling by using `bbox2`.

callobj

`callobj` draws a created object on the screen. Its argument *obj* takes the numeric identifier of the object you want to draw.

```
void callobj(obj)
Object obj;
```

You can use `callobj` to call one object from inside another. You can draw more complex pictures when you use a hierarchy of simple objects. For example, the program below uses a single `callobj (pearls)` to draw the object, a string of pearls, by calling the previously defined object `pearl` seven times.

```
Object pearl = 1, pearls = 2

makeobj(pearl);
    color(BLUE);
    for(angle=0; angle<3600; angle=angle+300) {
        rotate(300, 'y');
        circ(0.0, 0.0, 1.0);
    }
closeobj();
makeobj(pearls);
    for(i=0; i<7; i=i+1) {
        translate(2.0, 0.0, 0.0);
        color(i);
        callobj(pearl);
    }
closeobj();
```

Figure 16-2 is another example of using simple objects to build more complex ones. It defines a solar system as a hierarchical object. Calling one object `solarsystem` draws all the other objects named in its definition (the sun, the planets, and their orbits).

The system does not save global attributes before `callobj` takes effect. Thus, if an attribute, such as color, changes within an object, the change can affect the caller as well. You can use `pushattributes` and `popattributes` to preserve global attributes across `callobj`.

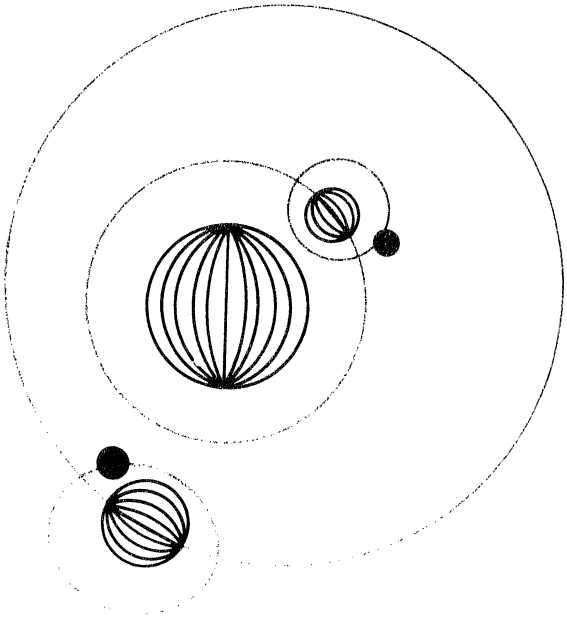
When you call a complex object, the system draws the whole hierarchy of objects in its definition. For example, in Figure 16-2, because the system draws the whole object `solarsystem` it can draw objects that are not visible in the viewport.

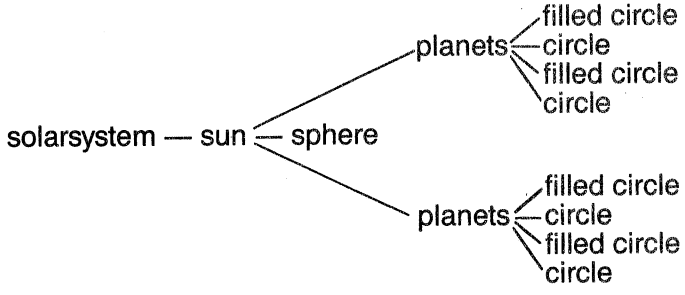
bbox2

`bbox2` determines whether or not an object is within the viewport, and whether it is large enough to be seen. It performs the graphical functions known as *pruning* and *culling*. Culling determines which parts of the picture are less than the minimum feature size, and thus too small to draw on the screen. Pruning calculates whether an object is completely outside the viewport.

`bbox2` takes as its arguments an object space bounding box (*x1*, *y1*, *x2*, *y2*) in coordinates, and minimum horizontal and vertical feature sizes (*xmin*, *ymin*) in pixels. The system calculates the bounding box, transforms it to screen coordinates, and compares it with the viewport. If the bounding box is completely outside the viewport, the routines between `bbox2` and the end of the object are ignored. If the bounding box is within the viewport, the system compares it with the minimum feature size. If it is too small in both the *x* and *y* dimensions, the rest of the routines in the object are ignored. Otherwise, the system continues to interpret the object.

```
void bbox2(xmin, ymin, x1, y1, x2, y2)
Screencoord xmin, ymin;
Coord x1, y1, x2, y2;
```

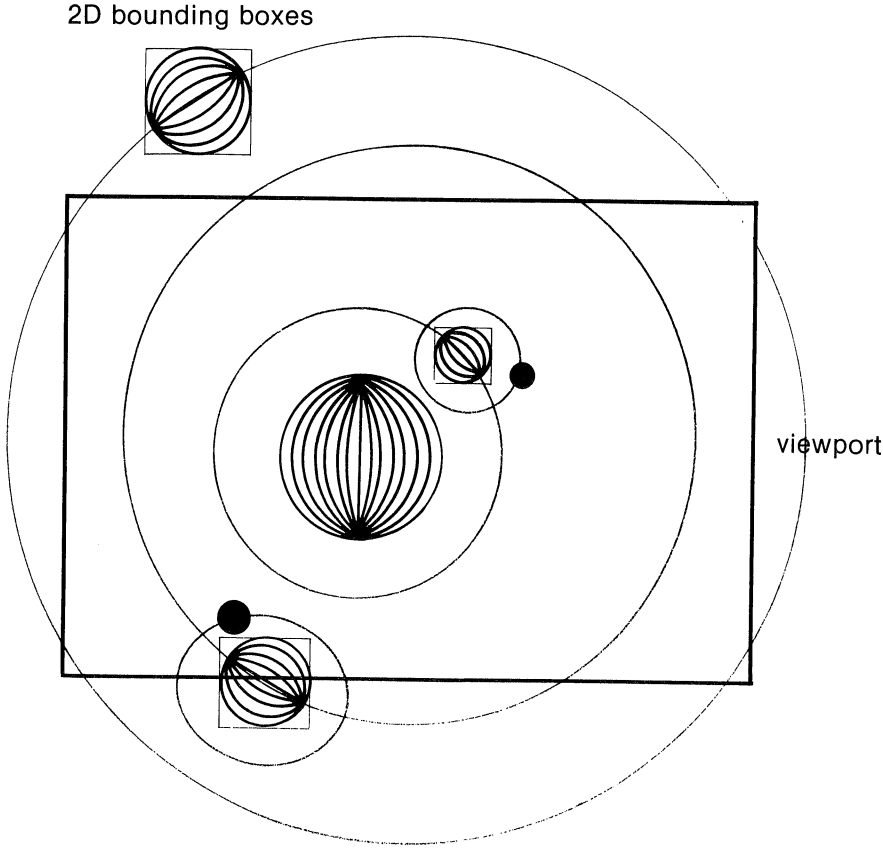




Solarsystem, a complex object, is defined hierarchically, as shown in the tree diagram. Branches in the tree represent *callobj* routines.

Figure 16-2. Defining a Hierarchical Object (solarsystem)

Figure 16-3 shows some of the objects within `solarsystem` juxtaposed on specified bounding boxes. The bounding boxes can perform pruning to determine which objects are partially in the viewport.



Bounding boxes are computed to determine which objects are outside the screen viewport. If the bounding box is entirely outside the viewport, the rest of the object display list is not traversed. The sphere in the bounding box that lies partially within the viewport is drawn and clipped to the edge of the viewport.

Figure 16-3. Bounding Boxes

16.3 Editing Objects

You can change an object by editing it. Editing requires that you identify and locate the drawing routines you want to change. You use two types of routines when you edit an object:

- *editing routines*, which add, remove, or replace drawing routines
- *tag routines*, which identify locations of drawing routines within an object

If you have to edit graphical objects frequently, you should build your own custom data structures and traversal routines, rather than use graphical objects. The editing routines that follow are best suited for infrequent and simple editing operations.

editobj

To open an object for editing, use `editobj`. A pointer acts as a cursor that appends new routines. The pointer is initially set to the end of the object. The system appends graphics routines to the object until either a `closeobj` or a pointer positioning routine `objdelete`, (`objinsert`, or `objreplace`) executes.

The system interprets the editing routines following `editobj`. Use `closeobj` to terminate your editing session. If you specify an undefined object, an error message appears.

```
void editobj(obj)
Object obj;
```

getopenobj

To determine if any object is open for editing, use `getopenobj`. If an object is open, it returns the object's numeric identifier. It returns -1 if no object is open.

```
Object getopenobj()
```

16.3.1 Using Tags

Tags locate items within a graphical object that you want to edit. Editing routines require tag names as arguments. `STARTTAG` is a predefined tag that goes before the first item in the list; it marks the beginning of the list. `STARTTAG` does not have any effect on drawing or modifying the object. Use it only to return to (find) the beginning of the list. `ENDTAG` is a predefined tag that is positioned after the last item on the list; it marks the end of the list. Like `STARTTAG`, `ENDTAG` does not have any effect on drawing or modifying the object. Use it to find the end of the graphical object. When you call `makeobj` to create a list, `STARTTAG` and `ENDTAG` automatically appear. You cannot delete these tags. When an object is opened for editing, there is a pointer at `ENDTAG`, just after the last routine in the object. To perform edits on other items, refer to them by their tags.

maketag

You can use tags to mark items you might want to change. You explicitly tag routines with `maketag`. You specify a 31-bit numeric identifier and the system places a marker between two items. You can use the same tag name in different objects.

```
void maketag(t)
Tag t;
```

newtag

`newtag` also adds tags to an object, but uses an existing tag to determine its relative position within the object. `newtag` creates a new tag that is offset beyond the other tag by the number of lines given in its argument *offst*.

```
void newtag(newtg, oldtg, offst)
Tag newtg, oldtg;
long offst;
```

istag

`istag` tells whether a given tag is in use within the current open object. `istag` returns TRUE if the tag is in use, and FALSE if it is not. The result is undefined if there is no currently open object.

```
Boolean istag(t)
Tag t;
```

gentag

`gentag` generates a unique integer to use as a tag within the current open object.

```
Tag gentag()
```

deltag

`deltag` deletes tags from the object currently open for editing. Remember, you cannot delete the special tags `STARTTAG` and `ENDTAG`.

```
void deltag(t)
Tag t;
```

16.3.2 Inserting, Deleting, and Replacing within Objects

The routines `objinsert`, `objdelete`, and `objreplace` allow you to add, delete, or replace routines in a graphical object.

objinsert

Use `objinsert` to add routines to an object at the location specified in *t*. `objinsert` takes a tag as an argument, and positions an editing pointer on that tag. The system inserts graphics routines immediately after the tag. To terminate the insertion, use `closeobj` or another editing routine (`objdelete`, `objinsert`, `objreplace`).

```
void objinsert(t)
Tag t;
```

objdelete

`objdelete` removes routines from the current open object. It removes everything between *tag1* and *tag2*—it deletes routines and other tag names. For example, `objdelete(STARTTAG, ENDTAG)` would delete every routine. The system ignores `objdelete` if no object is open for editing. This routine leaves the pointer at the end of the object after it executes.

```
void objdelete(tag1, tag2)
Tag tag1, tag2;
```

objreplace

`objreplace` combines the functions of `objdelete` and `objinsert`. It provides a quick way to replace one routine with another that occupies the same amount of space in the graphical object. Its argument is a single tag, *t*. Graphics routines that follow `objreplace` overwrite existing routines until a `closeobj` or editing routine (`objinsert`, `objreplace`, `objdelete`) terminates the replacement.

Note: `objreplace` requires that the new routine to be exactly the same length in characters as the previous one. Use `objdelete` and `objinsert` for more general replacement.

```
void objreplace(t)
Tag t;
```

Example—Editing an Object

The following is an example of object editing. First, the object `star` is defined:

```
makeobj(star);
    color(GREEN);
    maketag(BOX);
    recti(1, 1, 9, 9);
    maketag(INNER);
    color(BLUE);
    poly2i(8, Inner);
    maketag(OUTER);
    color(RED);
    poly2i(8, Outer);
    maketag(CENTER);
    color(YELLOW);
    pnt2i(5, 5);
closeobj();
```

Then, `star` is edited with the following routine to give a modified object:

```
editobj(star);
    circi(1, 5, 5);
    objinsert(BOX);
    recti(0, 0, 10, 10);
    objreplace(INNER);
    color(GREEN);
closeobj();
```

The object resulting from the editing session is equivalent to an object created by the following code:

```
makeobj(star);
    color(GREEN);
    maketag(BOX);
    recti(0, 0, 10, 10);
    recti(1, 1, 9, 9);
    maketag(INNER);
    color(GREEN);
    poly2i(8, Inner);
    maketag(OUTER);
    color(RED);
    poly2i(8, Outer);
    maketag(CENTER);
    color(YELLOW);
    pnt2i(5, 5);
    circi(1, 5, 5);
closeobj();
```

16.3.3 Managing Object Memory

Editing can require large amounts of memory. `compactify` and `chunksize` perform memory management tasks.

compactify

As memory is modified by the various editing routines, an open object can become fragmented and be stored inefficiently. When the amount of wasted space becomes large, the system automatically calls `compactify` during the `closeobj` operation. This routine allows you to perform the compaction explicitly. Unless you insert new routines in the middle of an object, compaction is not necessary.

Note: `compactify` uses a significant amount of computing time. Do not call it unless the amount of available storage space is critical; use it sparingly when performance is a consideration.

```
void compactify(obj)
Object obj;
```

chunksize

`chunksize` lets you specify the minimum chunk of memory necessary to accommodate the largest GL command you want to call. Normally, this is a call to `poly` or `polf` with a large number of vertices. If there is a memory shortage, you can use `chunksize` to allocate memory differently to an object. `chunksize` specifies the minimum amount of memory that the system allocates to an object. The default *chunk* is 1020 bytes. When you specify *chunk*, its size should vary according to the needs of the application. As the object grows, more memory is allocated in units of size *chunk*. You call `chunksize` only once after `winopen`, and before the first `makeobj`.

`chunksize` can help you use memory economically. For example, if you are using graphical objects that require very little memory, you can use the system more efficiently by specifying smaller chunks of memory. There are drawbacks to the use of `chunksize`. There is both memory and execution time overhead associated with each chunk; for this reason, using many small chunks can be inefficient.

```
void chunksize(chunk)
long chunk;
```

16.4 Mapping Screen Coordinates to World Coordinates

mapw

`mapw` takes a 2-D screen point and maps it onto a line in 3-D world space. Its argument *vobj* contains the viewing, projection, and viewport transformations that map the current displayed objects to the screen.

`mapw` reverses these transformations and maps the screen coordinates back to world coordinates. It returns two points (*wx1*, *wy1*,*wz1*) and (*x2*, *wy2*, *wz2*), which specify the endpoints of the line. The length of the line is arbitrary. *sx* and *sy* specify the screen point to be mapped.

```
void mapw(vobj, sx, sy, wx1, wy1, wz1, wx2, wy2, wz2)
Object vobj;
Screencoord sx, sy;
Coord *wx1, *wy1, *wz1, *wx2, *wy2, *wz2;
```

mapw2

`mapw2` is the 2-D version of `mapw`. In 2D, the system maps screen coordinates to world coordinates rather than to a line. Again, *vobj* contains the projection and viewing transformations that map the displayed objects to world coordinates; *sx* and *sy* define screen coordinates. *wx* and *wy* return the corresponding world coordinates. If the transformations in *vobj* are not 2D (i.e., not orthogonal projections), the result is undefined.

```
void mapw2(vobj, sx, sy, wx, wy)
Object vobj;
Screencoord sx, sy;
Coord *wx, *wy;
```

17. Feedback

Note: Feedback is different on each IRIS-4D Series system. Avoid using the feedback mechanism unless it is absolutely necessary.

Feedback is a system-dependent mechanism that uses the Geometry Pipeline to do calculations and to return the results of those calculations to the user process. From a hardware point of view, the net result of most Graphics Library calls is to send a series of commands and data down the Geometry Pipeline. In the pipeline, points are transformed, clipped, and scaled; lighting calculations are done and colors computed; the points, lines and polygons are scan-converted; and finally, pixels in the bitplanes are set to the appropriate values.

When the system is put into feedback mode, the Graphics Library commands send exactly the same information into the front of the graphics pipeline, but the pipeline is short-circuited, and the results of some of the calculations are returned before the standard drawing process is complete. The pipeline can be broken down into many distinct stages, the first of which is composed of Geometry Engines. The Geometry Engines transform, clip, and scale vertices to screen coordinates, and do the basic lighting calculations. In feedback mode, the raw output from the Geometry Engines is sent back to the host process, and no further calculations are done.

The hardware that makes up the Geometry Engine subsection of the pipeline is different on all Silicon Graphics systems. The command and data format differs and certain calculations are done on some systems and not on others. In spite of object code compatibility, the results of feedback are not compatible. If you use feedback, your code must be written differently for every system, and each time a new system is introduced, it will probably have to be modified.

Almost all feedback-type calculations can easily be done in portable host software. There are, however, a few places where feedback might be valuable. If you have code that draws an object on the screen, and you would like to draw the same picture on a plotter with a different resolution than that of the screen, you can change just the `viewport` subroutine (which controls the scaling of coordinates) so that it scales to your plotter coordinates, and then draw the picture in feedback mode. The transformed data returned to your process can often be interpreted and used to drive a plotter. `feedback` puts the system into feedback mode, and any set of graphics subroutines can then be issued, followed by `endfeedback`. All the commands and data that come out of the Geometry Engine subsection as a result are stored in a buffer supplied when the initial call to `feedback` was made.

The following IRIS-4D/GT/GTX program transforms some simple geometric figures and the results are returned in a buffer. The program does no interpretation—it simply prints out the contents of the buffer. The interpretation of the feedback data is covered later in this chapter; this example simply illustrates the mechanism for getting into and out of feedback mode.

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

float vert[3][2] = {{1.0, 2.0}, {3.0, 4.0}, {2.0, 8.0}};

main()
{
    short feedbuf[100];
    long i, count;
```

```
    winopen("feedback");
    feedback(feedbuf, 100);
    bgnpoint();
    v2f(&vert[0][0]);
    v2f(&vert[1][0]);
    v2f(&vert[2][0]);
    endpoint();
    bgnpolygon();
    v2f(&vert[0][0]);
    v2f(&vert[1][0]);
    v2f(&vert[2][0]);
    endpolygon();
    count = endfeedback(feedbuf);
    for (i = 0; i < count; i = i + 1)
   printf("%d ", feedbuf[i]);
}
```

In this example, `feedback` puts the system to into feedback mode, and tells the system to return all data in the buffer named *feedbuf*. In addition, the 100 indicates that the size of the buffer is 100 data items. If more than 100 items of data are generated, only the first 100 are saved. The geometry is drawn (in feedback mode), and `endfeedback` ends the feedback session. `endfeedback` returns the total number of items returned in the buffer. If an overflow occurs, the system returns 100. Finally, the loop at the end prints out the contents of the feedback buffer.

After a feedback session, the feedback buffer can contain any or all of the following data: points, lines, moves, draws, polygons, character move, `passthrough`, z-buffer, `linestyle`, `setpattern`, `linewidth`, and `lsrepeat` values.

On the IRIS-4D/VGX and Personal IRIS, `feedback` returns 32-bit floating point values instead of 16-bit integers. On all other IRIS-4D Series systems, `feedback` returns 16-bit integers.

In feedback mode, all the graphical subroutines are transformed, clipped, and scaled by the viewport, and all lighting calculations are done. Because of

clipping, more or fewer vertices might appear in the feedback buffer than were sent in. For example, a point might either make it through or be clipped out. A line drawn by the sequence:

```
bgnline();
    v3f(vert1);
    v3f(vert2);
    v3f(vert3);
endline();
```

can come out as nothing, as a move-draw, as a move-draw-draw, or as a move-draw-move-draw. A three-sided polygon can come out with up to nine sides, due to clipping against all six canonical clipping planes (see Figure 17-1), even more sides if user-defined arbitrary clipping planes are enabled. You can experiment with the effects of clipping on feedback by altering the `ortho2` parameters in the above example.

Because the length of the output is not generally predictable from the input, `passthrough` marks divisions in the input data. For example, if you send this sequence:

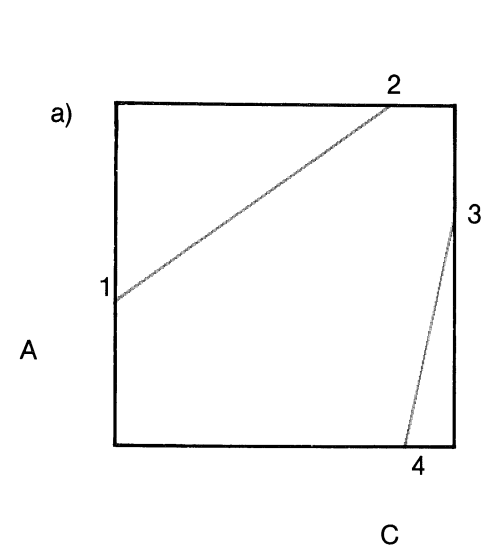
```
pnt(A);
passthrough(1);
pnt(B);
passthrough(2);
pnt(C);
passthrough(3);
pnt(D);
```

the parsed information in the feedback buffer might look like this:

```
transformed point (X)
passthrough (1)
passthrough (2)
transformed point (Y)
passthrough (3)
```

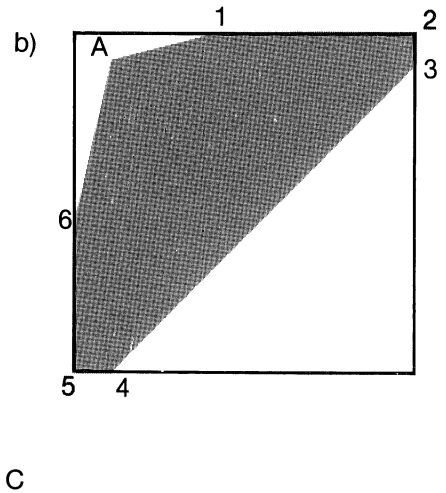
Point *X* is the transformed version of point *A*, and point *Y* is the transformed version of point *C*. Points *B* and *D* must have been clipped out.

The feedback data types are in the file *gl/feed.h* for your reference. All returned information is raw and system-specific.



The sequence: move(<A>)
draw(<B>)
draw(<C>)

becomes: move(<1>)
draw(<2>)
move(<3>)
draw(<4>)



The sequence: pmv (<A>)
pdr (<B>)
pdr (<C>)
pclos

becomes: pmv (<A>)
pdr (<1>)
pdr (<2>)
pdr (<3>)
pdr (<4>)
pdr (<5>)
pdr (<6>)
pclos

The clipping subsystem of the Geometry Pipeline can generate new routines; (a) shows a new *move* inserted in the routine stream while (b) shows a three-point polygon that turns into a seven-point polygon.

Figure 17-1. Effects of Clipping on Feedback

17.1 Feedback on IRIS-4D/GT/GTX Systems

Feedback data comes in groups of 8n+2 shorts, where *n* is the number of vertices involved:

Short #	Data
1	<data type>
2	<count>
3 thru (count+2)	<vertex data>

Table 17-1. IRIS-4D/G/GT/GTX Feedback Data

The vertex data is always arranged in groups of 8 (so count is a multiple of 8) and contain the values:

```
x, y, zhigh, zlow, r, g, b, alpha
```

x is the screen (not window) *x*-coordinate, *y*-1024 is the screen *y*-coordinate, *(zhigh<<16)+zlow* is the *z*-coordinate, and, *r*, *g*, *b*, and *alpha* are the red, green, blue, and alpha values.

By the time the data makes it through the geometry hardware, all the transformations have been done to it, including translations to put the data in the proper window. In the IRIS-4D/GT/GTX, the hardware screen *y*-coordinates begin at 1024 (for the bottom of the screen) and increase to 2047. Thus, 1024 must be subtracted to get what you would consider the screen *y*-coordinate.

24 bits of *z*-coordinate data is returned in two 16-bit chunks. The two chunks must be concatenated to get the full 24 bits of data. Finally, the red, green, blue, and alpha values are the colors that would be written into the frame buffer at the vertex. In RGB mode, all values vary between 0 and 255; in color map mode, the color index is sent as the red value and is in the range 0 to 4095. In color map mode, the values in the green, blue and alpha components are meaningless.

There are five possible kinds of data type: FB_POINT, FB_LINE, FB_POLYGON, FB_CMOV, and FB_PASSTHROUGH.

If three points come out of the Geometry Pipeline, the returned data consists of 26 shorts:

```
FB_POINT  24
x1, y1, zhigh1, zlow1, r1, g1, b1, alpha1
x2, y2, zhigh2, zlow2, r2, g2, b2, alpha2
x3, y3, zhigh3, zlow3, r3, g3, b3, alpha3
```

FB_LINE and FB_POLYGON are similar. FB_CMOV and FB_PASSTHROUGH always have 8 shorts of data as follows:

```
FB_CMOV 8
x1, y1, zhigh1, zlow1, r1, g1, b1, alpha1

FB_PASSTHROUGH 8
value, junk, junk, junk, junk, junk, junk, junk
```

17.2 Feedback on the Personal IRIS

The Personal IRIS has the following feedback tokens defined in *gl/feed.h*:

```
FB_POINT
FB_MOVE
FB_DRAW
FB_POLYGON
FB_CMOV
FB_PASSTHROUGH
FB_ZBUFFER
FB_LINESTYLE
FB_SETPATTERN
FB_LINEWIDTH
FB_LSREPEAT
```

Each group of feedback data begins with one of the above tokens to indicate data type. Vertex data for points, lines, and polygons always appears in groups of six floating-point values:

```
x, y, z, r, g, b
```

x and *y* are screen (not window) coordinates,*z* is the *z* value, and *r*, *g*, *b* are the red, green, and blue (RGB) values.

The RGB values are the colors that would be written into the frame buffer at the vertex. In RGB mode, all values vary between 0 and 255. In color map mode, the *r* value is the color index (between 0 and 4095) and the *g* and *b* values are ignored.

If a move, draw, or point (as in this example) comes out of the Geometry Pipeline, the returned data consists of seven floats:

```
FB_POINT
x, y, z, r, g, b
```

For polygons, feedback data includes a count number as well as the data type number. This number indicates how many of the next float values apply to the polygon. There are six for each vertex, so this number is always a multiple of six (6, 12, etc.).

For example, the returned data for a triangle consists of 20 floats:

```
FB_POLYGON   18.0
x1, y1, z1, r1, g1, b1
x2, y2, z2, r2, g2, b2
x3, y3, z3, r3, g3, b3
```

The 18.0 indicates three vertices with six values in each, and the 18 values follow it.

FB_CMOV returns only three floats of data:

```
FB_CMOV
x, y, z
```

The rest of the commands (FB_PASSTHROUGH, FB_ZBUFFER, FB_LINESTYLE, FB_SETPATTERN, FB_LINEWIDTH, FB_LSREPEAT) return only one float. For example, FB_PASSTHROUGH returns:

```
FB_PASSTHROUGH
value
```

17.3 Feedback on IRIS-4D/VGX Systems

The IRIS-4D/VGX returns 32-bit floating point numbers in feedback mode. The feedback data is in the following format:

```
<data type> <count> <count words of data>
```

There are five data types: FB_POINT, FB_LINE, FB_POLYGON, FB_CMOV, and FB_PASSTHROUGH. The actual values of these data types are defined in *gl/feed.h*. Following is the feedback format:

```
FB_POINT, count (9.0), x, y, z, r, g, b, a, s, t.

FB-LINE, count (18.0), x1, y1, z1, r1, g1, b1, a1, s1, t1,
x2, y2, z2, r2, g2, b2, a2, s2, t2.

FB_POLYGON, count (27.0), x1, y1, z1, r1, g1, b1, a1, s1,
t1, x2, y2, z2, r2, g2, b2, a2, s2, t2, x3, y3, z3, r3, g3,
b3, a3, s3, t3.

FB_PASSTHROUGH, count (1.0), passthrough.

FB-CMOV, count (3.0), x, y, z.
```

The *x* and *y* values are in floating point screen coordinates, the *z* value is the floating point transformed *z*. Red, green, blue, and alpha are floating point values ranging from 0.0 to 255.0 in RGB mode. In color map mode, the color index is stored in the red value and ranges from 0.0 to 4095.0. The green, blue, and alpha values are undefined in color map mode. The *s* and *t* values are in floating point texture coordinates.

17.4 Additional Notes on Feedback

Any graphics subroutines can be called between `feedback` and `endfeedback`, but only subroutines generating points, lines, polygons, cmovs, or passthroughs can generate values in the feedback buffer. If, for example, you are writing code to generate both a display and data for a plotter, certain data can be lost (polygon patterning, for example). If it is necessary to use this information in the plotting package, you should encode it somehow into `passthrough` commands.

Also note that subroutines such as `curve`, `patch`, and `mesh`, generate feedback buffer data, because they are converted in the graphics pipeline into a series of lines or polygons.

18. Textures

This chapter introduces the texture mapping features of the Graphics Library. Currently these features are available only on IRIS-4D/VGX systems. If you are not using a VGX system, you might want to skip this chapter.

Texture mapping is a powerful visualization aid that can efficiently add detail and realism to a raster image. A *texture* is a function defined in *texture space* that is warped, or *mapped*, by a specified mapping into an *object space*. Although the texture function, or texture for short, can be an arbitrarily dimensioned array of data, this chapter focuses on 2-D images, the most commonly used textures. Similarly, this chapter considers mappings only onto geometric primitives in 3-D space, namely polygons, lines, and points.

Texture mapping is a general technique. In a scientific visualization setting, it can simulate physical properties of objects. For example, a 1-D image of temperature represented by color can be mapped onto an object to illustrate thermal gradients. For photorealistic applications, texture mapping can simulate a wide variety of lighting effects, including specular and diffuse reflections, transparency, and shadows. A common and illustrative use of texturing is to map surface color and patterns to enhance visual realism. For example, a 2-D image of wood grain wrapped around a rectangular solid can generate a realistic-looking two-by-four board. You can create patterned surfaces such as brick walls and fabrics by repeating textures across a surface.

This chapter concentrates on the three major functional blocks that constitute the texture mapping process (Figure 18-1)—texture coordinates, texture function, and texture environment.

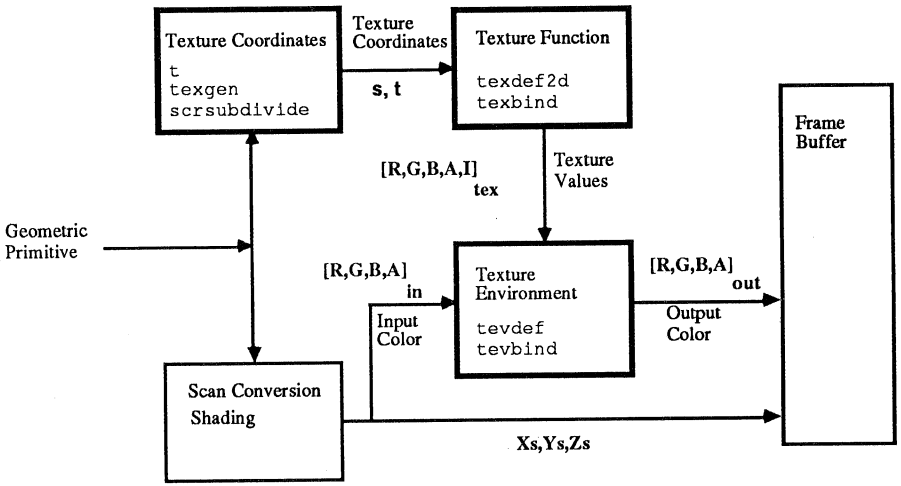


Figure 18-1. Functional Overview of Texture Mapping

A simplified description of the texture mapping process is:

1. First you assign texture coordinates to vertices to define a mapping from texture space to geometry in object space. Section 18.1 describes the mechanisms for assigning texture coordinates and the methods for interpolating them to produce texture coordinates at pixels.

2. Next the GL produces texture values from the texture coordinates according to the texture function. This step, which includes sampling and filtering issues, is discussed in Section 18.2.

3. Finally, the texture environment determines final color from input values produced by shading and the texture function. Section 18.3 describes this.

Section 18.4 concludes the chapter with a discussion on strategies for achieving the maximum texture mapping performance from the Graphics Library.

To illustrate the mechanics of texture mapping using the Graphics Library, refer to the following program, which replicates a brick pattern across a quadrilateral. A more complex example program is included at the end of this chapter. It illustrates the use of multiple textures, including a 1-D texture that uses texture alpha for blending.

Program Example 1

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

float texprops[] = {TX_MINFILTER, TX_POINT,
                    TX_MAGFILTER, TX_POINT,
                    TX_WRAP, TX_REPEAT, TX_NULL};

/* Texture color is brick-red */
float tevprops[] = {TV_COLOR, .75, .13, .06, 1.,
                    TV_BLEND, TV_NULL};

/* Subdivision parameters */
float scrparams[] = {0., 0., 10.};

unsigned long bricks[] =          /* Define texture image */
    {0x00ffffff, 0xffffffff,
     0x00ffffff, 0xffffffff,
     0x00ffffff, 0xffffffff,
     0x00000000, 0x00000000,
     0xffffffff, 0x00ffffff,
     0xffffffff, 0x00ffffff,
     0xffffffff, 0x00ffffff,
     0x00000000, 0x00000000};

/* Define texture and vertex coordinates */
float t0[] = {0., 0.},    v0[] = {-2., -4., 0.};
float t1[] = {16., 0.},   v1[] = {2., -4., 0.};
float t2[] = {16., 32.}, v2[] = {2., 4., 0.};
float t3[] = {0., 32.},   v3[] = {-2., 4., 0.};

main()
{
    if (getgdesc(GD_TEXTURE) == 0) {
        fprintf(stderr, "texture mapping not available on
                                this machine\n");
        return 1;
    }
```

```
    keepaspect(1, 1);
    winopen("brick");
    subpixel(TRUE);
    RGBmode();
    lsetdepth(0x0, 0x7fffff);
    doublebuffer();
    gconfig();

    mmode(MVIEWING);
    ortho(-4., 4., -4., 4., 1., 16.);

    texdef2d(1, 1, 8, 8, bricks, 0, texprops);
    tevdef(1, 0, tevprops);
    texbind(TX_TEXTURE_0, 1);
    tevbind(TV_ENV0, 1);

    scrsubdivide(SS_OFF, scrparams);/* Screen subdivision */

    translate(0., 0., -6.); /* Move poly away from viewer */

    while(!getbutton(LEFTMOUSE)) {
        cpack(0x0);
        clear();

        pushmatrix();
        rotate(getvaluator(MOUSEX)*5,'y');
        rotate(getvaluator(MOUSEY)*5,'x');

        cpack(0xffcccccc);          /* Cement color */
        bgnpolygon();          /* Draw textured rectangle */

        t2f(t0);    v3f(v0);
        t2f(t1);    v3f(v1);
        t2f(t2);    v3f(v2);
        t2f(t3);    v3f(v3);
        endpolygon();
        popmatrix();
        swapbuffers();
    }

    texbind(TX_TEXTURE_0, 0);       /* Turn off texturing */
}
```

18.1 Texture Coordinates—t, texgen, scrsubdivide

This section describes how to map textures onto geometry using texture coordinates and how texture coordinates are generated at screen pixels. In Figure 18-2, coordinate axes *S* and *T* define 2-D texture space. By definition, a texture lies in the range 0 to 1 along both axes. *(s,t)* pairs not necessarily limited to this range index the texture.

To define a mapping, assign texture coordinates to the vertices of a geometric primitive; this process is called *parameterization.* Figure 18-2 shows a parameterized triangle and how the parameterization defines the mapping between coordinate systems. You can either assign texture coordinates explicitly with the `t` subroutine, or you can let the system automatically generate and assign texture coordinates using the `texgen` subroutine.

Next, the current texture matrix transforms the texture coordinates. This matrix is set while in `mmode (MTEXTURE)` and is a standard 2-D transformation matrix. Thus, the *(s,t)* pair is treated as a 2-D point and is transformed accordingly.

The final step generates *s* and *t* at every pixel center inside a geometric primitive by interpolating between the vertex texture coordinates during scan-conversion. The IRIS-4D/VGX uses hardware to linearly interpolate texture coordinates. Although hardware interpolation is very fast, it is incorrect for perspective projections. The `scrsubdivide` subroutine improves *s* and *t* interpolation—and consequently image quality—for perspective projections.

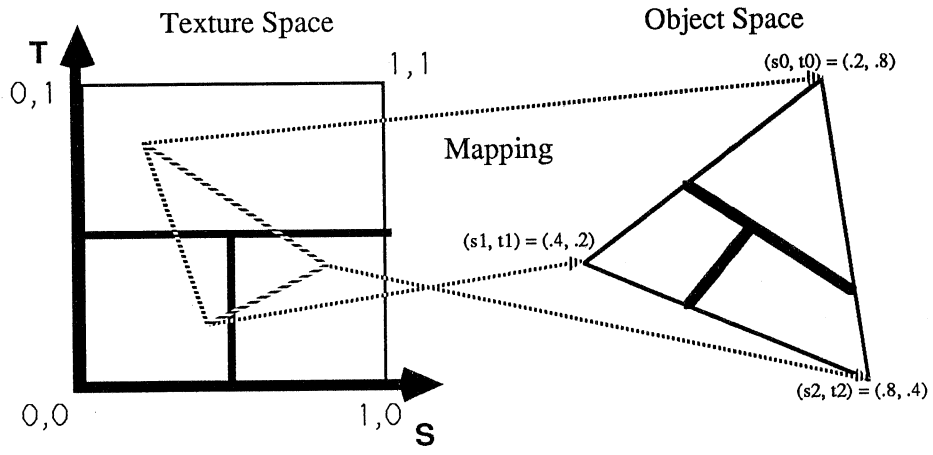


Figure 18-2. Mapping from Texture Space to Geometry in Object Space

t

The `t` subroutines specify texture coordinates. Currently supported forms take a two-element array whose type can be short, long, float, or double (Table 18-1). The first array component is *s*, the second is *t*.

Array Type	2-D Form
short integer	t2s
long integer	t2i
float	t2f
double	t2d

Table 18-1. The t Subroutine

In the sample program, notice that a `t` subroutine is invoked before each `v` subroutine to assign different texture coordinates to each vertex. This particular parameterization defines a simple mapping that preserves the rectangular shape of the brick texture. Because the *s* and *t* values are greater than 1, the texture is repeated.

Although the given mapping is straightforward, you can stretch, squash, or otherwise distort the texture as it is mapped onto the polygon by changing the texture coordinates and/or polygon geometry. For example, change the *t3* texture coordinate definition in the sample program to `float t3[] = {0., 8.};` note how the brick pattern is stretched in one half of the polygon.

texgen

`texgen` generates texture coordinates as a function of object geometry. Current generation algorithms compute the distance of a vertex from a specified plane and calculate texture coordinates proportional to this distance. The `TG_LINEAR` mode defines the plane in object coordinates so the parameterization is fixed with respect to object geometry. For example, you can use this mode to texture map terrain using sea level as the reference plane. In this case, the altitude of a terrain vertex is its distance from the reference plane. You can use `TG_LINEAR` so that vertex altitude indexes the texture to map white snow onto peaks and green grass onto foothills.

The `TG_CONTOUR` mode defines the specified plane in eye coordinates. The ModelView matrix in effect at the time of mode definition transforms the plane equation. Thus, the transformation matrix is not necessarily the same as that applied to vertices. This mode establishes a "field" of texture coordinates that can produce dynamic contour lines on moving objects.

Coordinates are generated on a per-vertex basis and override coordinates specified by the `t` commands. You can independently control the generation of either or both texture coordinates. If you generate only one coordinate, the other is specified by the `t` subroutines.

The form of the plane equation used is $Ax + By + Cz + D = 0$ and is represented by the values A, B, C, and$D$. In this formulation, the plane normal is the vector $[A,B,C]$ and the plane constant is D. For example, the plane $X=Y$ is defined by {1., -1., 0., 0.}.

The following code fragment illustrates how to use the `TG_LINEAR` function to generate s coordinates proportional to vertex distance from the object coordinate plane, $X=Y$. The first call to `texgen` defines the generation algorithm for the s coordinate; the second call activates s coordinate generation so that the system generates an s coordiante for each vertex..

```
float tgparams[] = {1., -1., 0., 0.};
texgen(TX_S, TG_LINEAR, tgparams);
texgen(TX_S, TG_ON, tgparams);
```

scrsubdivide

On the IRIS-4D/VGX, texture coordinates are linearly interpolated in screen space by hardware the same way color is interpolated. Although this produces fast rendering, it is mathematically incorrect for perspective projections. For example, you can modify the sample program by replacing the `ortho` subroutine with `perspective(600, 1., 1., 16.)`, which introduces perspective distortion. Because of incorrect interpolation, textures no longer appear fixed to a surface but shift as the surface moves. This effect is called "swimming."

Swimming occurs because texture coordinates are interpolated after the perspective division (in screen coordinates) when they should be interpolated in eye coordinates. Because the hardware does not support eye coordinate interpolation, you can use screen subdivision, `scrsubdivide`, to improve texture coordinate interpolation. Screen subdivision can also improve the accuracy of fog by correctly interpolating w (see Chapter 13).

The `SS_DEPTH` algorithm subdivides polygons and lines into smaller pieces. Colors, texture coordinates, and the homogeneous coordinate w at newly generated vertices are correctly interpolated in eye coordinates rather than in screen coordinates. Because incorrect interpolation is limited to smaller pieces, error globally decreases and image quality increases. Consequently, you can "tune" image quality by modifying the amount of subdivision.

Note: `scrsubdivide` is most effective for large, non-tessellated polygons and lines. Highly tessellated surfaces (e.g., curved surfaces) have, in essence, already been subdivided and thus benefit little from further subdivision.

SS_DEPTH subdivision slices screen coordinate polygons and lines by a fixed grid in *Z*. Spacing between *Z* planes is constant throughout the grid and is determined by the three `scrsubdivide` parameters: maximum screen *Z*, minimum screen size, maximum screen size. The first value in the parameter list specifies the desired distance between subdivision planes in units set by `lsetdepth`. If polygon slices generated using this metric span a distance in pixels less than minimum screen size, the distance between subdivision planes is increased until the slices are larger than the minimum screen size. This can occur when a polygon is oriented edge-on, so that it spans little screen distance. If polygon slices generated using the maximum screen *Z* metric span a distance in pixels greater than maximum screen size, the distance between subdivision planes is decreased until the slices are smaller than the maximum screen size. This parameter is often useful for polygons with little perspective that suffer from too little subdivision.

In practice, the minimum and maximum screen size parameters are used to keep slices from becoming too small or too big, respectively. However, these parameters can introduce situations where polygons that share an edge are sliced by differently spaced grids. This generates *T-vertices* that can cause pixel dropout along the shared edges. To avoid T-vertices, you can "turn off" the screen size parameters by setting them to 0 so only the maximum screen *Z* parameter is used. You can turn off any parameter by setting it to 0. For example, a parameter list of {0., 0., 10.} specifies subdivision every 10 pixels.

The following code fragment illustrates how the sample program uses screen subdivision:

```
float scrparams[] = {0., 0., 10.};
scrsubdivide(SS_OFF, scrparams);
```

To turn on screen depth subdivision, change the SS_OFF mode to SS_DEPTH. With the parameter list of {0., 0., 10.}, the quadrilateral is subdivided every 10 pixels and the image quality is improved. You can view the tessellation produced by `scrsubdivide` by drawing only polygon outlines using the `polymode(PYM_LINE)` subroutine (see Chapter 2).

18.2 Texture Functions—texdef2d, texbind

The previous section addressed how texture coordinates are assigned to and interpolated across geometric primitives. This section discusses how texture values are computed from given texture coordinates. How texture values are used to modify the color and opacity of a pixel is the subject of Section 18.3, where `tevdef` and `tevbind` are discussed.

Note: All geometry including polygons, lines, points, and characters are texture mapped. Characters always have texture coordinates (0.,0.).

This section introduces texture function—a function that returns one or more values for a given texture coordinate. The GL subroutine `texdef2d` defines such a function. The subroutine `texbind` activates a texture function. The `texdef2d` and `texbind` combination is similar to the lighting and material subroutines, `lmdef` and `lmbind`. `texdef2d` and `lmdef` both define functions and assign a unique index for each function defined. An optional set of attributes is associated with each function. Texture functions can be redefined by calling `texdef2d` with the index of a previously defined function. As with materials, only one texture function can be active, or bound, at a time. The binding process and defining process are separated for performance reasons—it takes substantially less time to activate a texture than to define one.

A texture function consists of an image defined as a 2-D array of pixels with one to four components per pixel, and a set of properties that determine how samples are derived from the image. Regardless of the pixel dimensions, the image is mapped into an *st*-coordinate range such that its lower-left corner is (0.,0.) and its upper-right corner is (1.,1.). The texture function's property set determines how the texture image is sampled and how the texture function evaluates outside the range (0.,0.) , (1.,1.).

For each pixel to be textured, the texture function generates texture values based on the texture coordinates of the pixel's center, and the area in texture pixels onto which the pixel maps.

The brick texture function in the example program is created with the following `texdef2d` subroutine:

```
float texprops[] = {TX_MINFILTER, TX_POINT,
                    TX_MAGFILTER, TX_POINT,
                    TX_WRAP,TX_REPEAT, TX_NULL};

texdef2d(1, 1, 8,  8, bricks, 0, texprops);
```

The brick texture is activated using the `texbind` call:

```
texbind(TX_TEXTURE_0, 1);
```

The texture image array format is consistent with `lrectread`. The image is a packed array of pixels stored as an array of `unsigned long` words. Each row of pixels starts `long` word aligned, so rows must be byte-padded as necessary. The image array has one to four components per pixel and is stored in the following format:

Components	Pixel Type	Byte Ordering
1-Component	Intensity	I0, I1, I2, I3, I4, ...
2-Component	Intensity-Alpha	A0, I0, A1, I1, A2, ...
3-Component	Red, Green, Blue	B0, G0, R0, B1, G1, ...
4-Component	Red, Green, Blue, Alpha	A0, B0, G0, R0, A1, ...

Table 18-2. Texture Image Array Format

In the `texdef2d` subroutine, the first parameter specifies the unique index, or name, that identifies the texture. The second parameter specifies the number of 8-bit components per texture pixel and the corresponding number of 8-bit components that each texture sample produces. The third and fourth parameters specify the width and height, in pixels, of the texture image. The fifth parameter is a pointer to the image array. The last two parameters specify a list of optional properties to modify the texture function.

The optional parameter list is an array of symbols terminated with the TX_NULL symbol. In this example, texprops explicitly specifies TX_MINFILTER, the filter function for minifying texture (used whenever the pixel being textured maps to an area greater than one square texture pixel), and TX_MAGFILTER, the filter function used when the pixel being textured maps to an area less than or equal to one texture pixel. (See Section 18.2.1 for a discussion of minification and magnification filters.) In this example, TX_MINFILTER and TX_MAGFILTER are set to use point sampling. TX_WRAP, which specifies what to do when *st* coordinates are outside the range (0.,1.), is set to TX_REPEAT, which specifies that only the fractional parts of the texture coordinates are used, thereby creating a repeating pattern.

By setting TX_WRAP to TX_REPEAT, the small 8x8 pattern is repeated across the polygon, creating an entire wall of bricks. If you replace TX_REPEAT with TX_CLAMP, you would see the brick pattern only once, in the corner of the polygon, where the *s* and *t* coordinates are in the range (0.,1.). The edges of the texture would be smeared across the rest of the polygon. TX_CLAMP is useful for preventing wrapping artifacts when mapping a single image onto an object.

18.2.1 Minification and Magnification Filters

During the texture mapping process, the texture function computes texture values based on the texture coordinates at the center of a pixel being textured and the area in texture space onto which the pixel maps. One of two filtering algorithms is used depending on the size of this area. If the area is less than the area of one texture pixel, the texture is magnified at this pixel, and the texture function's magnification algorithm is used. Conversely, if the area is greater than the area of one texture pixel, the texture is minified at this pixel, because the pixel being textured maps onto more than one texture pixel. In this case the texture function's minification filter is used.

Computing filtered samples or values from the texture function is computationally the most expensive part of the texture mapping process. The GL provides a variety of minification and magnification filters from which to choose. The filter performances vary in terms of speed and image quality. There is not a minification/magnification filter pair that is ideal for all applications; it is wise to experiment with various combinations. The following filters are available:

TX_MAGFILTER	TX_MINFILTER
TX_POINT	TX_POINT
TX_BILINEAR	TX_BILINEAR
	TX_MIPMAP_POINT
	TX_MIPMAP_LINEAR
	TX_MIPMAP_BILINEAR

Table 18-3. texdef2d Filter Choices

There are two magnification filters to choose from: TX_POINT and TX_BILINEAR. TX_POINT returns the value of the texture pixel that maps the closest to the center of the pixel being textured. TX_BILINEAR returns the weighted average of the four texture pixels that map the closest to the center of the pixel. TX_POINT is faster than TX_BILINEAR, but has the drawback that mapped textures can appear boxy, as there is not as smooth a transition between texture pixels as there is with TX_BILINEAR. If the texture image does not have sharp edges, this effect might not be noticeable.

The TX_POINT or TX_BILINEAR algorithms can be used for minification as well. The drawback of using one of these filters for minification is that only one or four of the texture pixels that map onto the area of the pixel being textured are considered in the texture value computation. If the texture is mapped so that it is shrunk by a factor greater than two, it might appear to shimmer as it moves or even appear to have a moire pattern on top of it. These aliasing artifacts result from undersampling, or not including in the texture value computation the contributions of all of the texture pixels that map onto the pixel being textured. The artifacts can be alleviated by using one of the MipMap filters.

The MipMap filters work with an array of prefiltered versions of the texture image called a MipMap. Each image in the array has half the resolution of the image before it, but still maps into the texture coordinate range (0.,0.) to (1.,1.). For any minification factor, there is one image in the MipMap with texture pixels that map to an area in texture space less than or equal to the area that the pixel being textured maps into. Samples interpolated from this image do not have the undersampling artifacts discussed above. The minification filters `TX_MIPMAP_POINT` and `TX_MIPMAP_BILINEAR` determine the correct MipMap image to sample from, then perform the same filtering computations as their non-MipMapped counterparts.

In applications where texture scaling varies widely across a polygon, the polygon can have samples from many different images of a MipMap, and the discrete transition between images of a MipMap can be distracting. In these situations, `TX_MIPMAP_LINEAR` might be the minification filter of choice. `TX_MIPMAP_LINEAR` linearly interpolates between point samples of the two images of the MipMap that have the closest scale factor, or area mapping, to the area that the pixel being textured maps into. The weighting of the interpolation is based on the relative closeness of the images' scale factors to the texture scale factor for the pixel being textured.

Sometimes you might not want the blurring from MipMap filtering, as is frequently the case when texture alpha is used to approximate geometry, such as in a row of trees. In these circumstances, `TX_BILINEAR` is a good minification filter choice. The textured object appears sharp at the cost of additional aliasing.

Both aliasing and blockiness of the textured polygon in the example program are reduced if you change the `texprops` array to:

```
float texprops[] = {TX_MINFILTER, TX_MIPMAP_BILINEAR,
                    TX_MAGFILTER, TX_BILINEAR,
                    TX_WRAP,TX_REPEAT, TX_NULL};
```

18.3 Texture Environments—tevdef, tevbind

A texture environment specifies how texture values modify the color and opacity of an incoming shaded pixel. The `tevdef` subroutine defines a texture environment, and the `tevbind` subroutine makes it active. As with `texbind`, there can be only one bound texture environment.

The texture environment function takes a shaded, incoming pixel color (*Rin,Gin,Bin,Ain*) and computed texture values (*tex*) as input, and outputs a new color (*Rout,Gout,Bout,Aout*).

There are three texture environment types: `TV_MODULATE`, `TV_BLEND`, and `TV_DECAL`. Each behaves differently based on the number of components the currently bound texture has.

TV_MODULATE Incoming color components are multiplied by texture values. Which texture value multiplies which incoming color component is a function only of the number of texture components (see `texdef`).

	Rout=	Gout=	Bout=	Aout=
1-component	Rin*Itex	Gin*Itex	Bin*Itex	Ain
2-component	Rin*Itex	Gin*Itex	Bin*Itex	Ain*Atex
3-component	Rin*Rtex	Gin*Gtex	Bin*Btex	Ain
4-component	Rin*Rtex	Gin*Gtex	Bin*Btex	Ain*Atex

TV_BLEND Texture values are used to blend the incoming color and the active texture environment color, which is a single RGBA constant (*Rconst,Gconst,Bconst,Aconst*). The texture environment color is specified with the TV_COLOR parameter.

	Blend Equations
1-component	Rout = Rin*(1-Itex) + Rconst*Itex Gout = Gin*(1-Itex) + Gconst*Itex Bout = Bin*(1-Itex) + Bconst*Itex Aout = Ain
2-component	Rout = Rin*(1-Itex) + Rconst*Itex Gout = Gin*(1-Itex) + Gconst*Itex Bout = Bin*(1-Itex) + Bconst*Itex Aout = Ain*Atex
3-component	not available
4-component	not available

TV_DECAL Texture alpha (referred to as *Atex* in the equations below) is used to blend the incoming color and the texture color.

	Blend Equations
1-component	not available
2-component	not available
3-component	Rout = Rtex Gout = Gtex Bout = Btex Aout = Ain
4-component	Rout = Rin*(1-Atex) + Rtex*Atex Gout = Gin*(1-Atex) + Gtex*Atex Bout = Bin*(1-Atex) + Btex*Atex Aout = Ain

In the example program, the texture environment is defined using the following call to `tevdef`:

```
float tevprops[] = {TV_COLOR, .75, .13, .06, 1.,
                    TV_BLEND, TV_NULL};
tevdef(1, 0, tevprops);
```

It is set to be the active texture environment using the following call to `tevbind`:

```
tevbind(TV_ENV0, 1);
```

The brick texture used is an intensity map. The texture environment creates a colored brick pattern by blending the gray polygon color and the red texture environment color based on the intensity of the brick texture map.

18.4 Texture Programming Hints

After you understand the basics of the Graphics Library's texture mapping routines, the following hints can be useful in getting the optimal performance from your IRIS-4D/VGX system.

scrsubdivide

Turn on `scrsubdivide` only when you need it

Because `scrsubdivide` generates many polygons from each incoming polygon, it is wise to turn off this feature when it is not needed, such as when you are drawing non-texture mapped polygons or highly tessellated texture mapped polygons.

Use only as much subdivision as you need

Choose the `scrsubdivide` parameters carefully. For maximum performance, only use as much subdivision as is necessary. Textures without high frequencies need less subdivision than those with high frequencies.

The GL resizes images when necessary

Internally, the Graphics Library works only with images whose dimensions are powers of two. `texdef2d` automatically resizes images as necessary. To avoid resizing, pass `texdef2d` images that have widths and heights that are powers of two.

Use as few components as necessary

The more components a texture has, the longer it takes to map the texture onto a polygon; for optimal speed, use as few components as possible. If you are not taking advantage of a texture's alpha, define the texture as a one- or three-component texture. If you do not need a full-color texture, define the texture with one or two components.

Use the simplest filter you need

The per-pixel speed of the texture filter functions is related to the number of interpolations the filter has to perform. The filters in order from fastest to slowest are : `TX_POINT`, `TX_MIPMAP_POINT`, `TX_MIPMAP_LINEAR`, `TX_BILINEAR`, and `TX_MIPMAP_BILINEAR`.

There is some overhead per polygon for using MipMap filters. If a scene has a large number of textured polygons, or if the polygons are subdivided finely, performance is improved if MipMap filters are not used.

High-resolution textures

Textures that exceed the maximum dimensions of the graphics hardware are resized to the maximum dimensions. The maximum dimensions for textures using MipMapping are half of those that do not. The effect is that large textures using MipMapping are fuzzier than those that do not.

Alphaless systems

Systems without alpha memory also lack storage for a fourth texture component. On such systems, the alpha component of four-component textures always appears to be 255. One-, two-, and three-component textures behave the same on systems with or without alpha.

`texdef2d` makes a copy of the texture image

The image passed to `texdef2d` is copied. This copy and all other data associated with the texture, such as its MipMap, are saved in the user's memory space until the texture is redefined or the program exits.

texbind

Bind textures as infrequently as possible

`texbind` can be a time-consuming operation, especially if the texture is not resident in the graphics hardware. To achieve maximum performance, draw all of the polygons using the same texture all together.

Texture caching

Hardware texture memory is a finite resource managed by the IRIX kernel. The kernel guarantees that the currently bound texture of a program resides in this memory, whenever the program owns the graphics pipe. Beyond that, the kernel keeps as many additional textures as possible in the hardware texture memory. 10-span VGX systems have 2.5 times as much texture memory as 5-span systems as long as the accumulation buffer is not used. When a texture is bound, if it is not resident in the texture memory and there is not enough room remaining for this texture, one or more of the resident textures are swapped out. To minimize the frequency of this swapping, use smaller textures or try to switch them less often.

General

Use `afunction` for fast drawing of objects with texture alpha

When the alpha component of a texture is used to approximate geometry (such as when a texture is used to describe a tree), the polygons must be blended into the scene in sorted order to properly realize the coverage defined by the alpha component. This sorting and blending requirement can be removed by using `afunction`. `afunction(0, AF_NOTEQUAL)` specifies that only pixels with non-zero alpha be drawn. See the `afunction` *man* page for more details.

Texture calls between bgn/end sequences

With the exception of the `t` commands, the texture subroutines described in this chapter cannot be called in the middle of `bgn/end` sequences such as `bgnpolygon/endpolygon`.

Turn off texturing when you are not texturing

A common bug in texturing applications is forgetting to turn off texturing when drawing non-textured objects. Not supplying texture coordinates does *not* disable texturing. Texturing is disabled only with one of the following subroutine calls: `texbind(TX_TEXTURE_0,0)` or `tevbind(TV_ENV0,0)`.

Texturing only works in RGB mode

The behavior of texturing is not defined in color index mode.

Program Example 2

The following code illustrates the use of multiple textures, including a 1-D texture that uses texture alpha for blending.

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

/* Texture environment */
float tevprops[] = {TV_MODULATE, TV_NULL};

/* 1D RGBA texture map representing temperature as color and
 * opacity */
float texheat[] = {TX_WRAP, TX_CLAMP, TX_NULL};
/* Black->blue->cyan->green->yellow->red->white */
unsigned long heat[] =   /* Translucent -> Opaque */
    {0x00000000, 0x55ff0000, 0x77ffff00, 0x9900ff00,
     0xbb00ffff, 0xdd0000ff, 0xffffffff};
```

```
/* Point sampled  1 component checkerboard texture */
float texbgd[] = {TX_MAGFILTER, TX_POINT,TX_NULL};
unsigned long check[] =
    { 0xff800000,       /* Notice row byte padding */
      0x80ff0000};

/* Subdivision parameters */
float scrparams[] = {0., 0., 10.};

/* Define texture and vertex coordinates */
float t0[] = {0., 0.}, v0[] = {-2., -4., 0.};
float t1[] = {.4, 0.}, v1[] = {2., -4., 0.};
float t2[] = {1., 0.}, v2[] = {2., 4., 0.};
float t3[] = {.7, 0.}, v3[] = {-2., 4., 0.};

main()
{

    if (getgdesc(GD_TEXTURE) == 0) {
        fprintf(stderr,
         "texture mapping not available on this machine\n");
        return 1;
    }
    keepaspect(1, 1);
    winopen("heat");
    subpixel(TRUE);
    RGBmode();
    lsetdepth(0x0, 0x7fffff);
    doublebuffer();
    gconfig();

    blendfunction(BF_SA,BF_MSA);     /* Enable blending */

    mmode(MVIEWING);
    perspective(600, 1, 1., 16.);

    /* Define checkerboard */
    texdef2d(1, 1, 2, 2, check, 0, texbgd);
    /* Define heat */
    texdef2d(2, 4, 7, 1, heat, 0, texheat);
    tevdef(1, 0, tevprops);
    tevbind(TV_ENV0, 1);
```

```
    translate(0., 0., -6.);

    while(!getbutton(LEFTMOUSE)) {
        cpack(0x00000000);
        clear();

        scrsubdivide(SS_OFF, scrparams);/* Subdivision off */
        texbind(TX_TEXTURE_0, 1);      /* Bind checkerboard */
        cpack(0xff102040);    /* Background rectangle color */
        bgnpolygon();           /* Draw textured rectangle */

        t2f(v0);     v3f(v0);     /* Notice vertex          */
        t2f(v1);     v3f(v1);     /* coordinates used       */
        t2f(v2);     v3f(v2);     /* as texture coordinates */
        t2f(v3);     v3f(v3);
        endpolygon();

        pushmatrix();
        rotate(getvaluator(MOUSEX)*5,'y');
        rotate(getvaluator(MOUSEY)*5,'x');

        /* Subdivision on */
        scrsubdivide(SS_DEPTH, scrparams);
        texbind(TX_TEXTURE_0, 2);              /* Bind heat */
        cpack(0xffffffff);  /* Heated rectangle base color */
        bgnpolygon();            /* Draw textured rectangle */
        t2f(t0);     v3f(v0);
        t2f(t1);     v3f(v1);
        t2f(t2);     v3f(v2);
        t2f(t3);     v3f(v3);
        endpolygon();

        popmatrix();
        swapbuffers();
    }

    texbind(TX_TEXTURE_0, 0);        /* Turn off texturing */
}
```

Appendix A. Scope of GL Statements

Each of the Graphics Library programming statements has a given scope within the IRIS-4D hardware or software. Some statements affect the state of the currently selected framebuffer, other statements affect the state of the current window, still others affect the state of the current process, etc. This appendix lists all the GL statements and defines the scope of each statement.

This appendix includes three tables. Tables A-1 and A-2 define codes that are used in Table A-3. Table A-1 defines the codes used to represent the state types of each GL programming statement, as well as the system resource on which the statement operates. This code is used in the leftmost column of Table A-3. Table A-2 defines codes used to convey additional information about each of the programming statements. Not every statement has a code as defined in Table A-2; when a statement does include one of these codes, it is in the rightmost column of Table A-3.

State types	Operates on
f	framebuffer (i.e. drawmode-dependent)
m	mmode dependent
c	colormap (there is a separate screen-wide color map for each framebuffer)
w	window
s	screen
p	process
t	textport (affects a different process from the caller's)
d	display (a collection of screens and input devices)
o	obsolete
r	renders into current framebuffer (selected by drawmode) or non-modal framebuffer state (such as texture coordinates, trimming curves)

Table A-1. GL State Types

The scope of each GL statement is limited according to the state types listed in Table A-1. These state types are used as abbreviations in the list of GL statements included in Table A-3.

Table A-2 lists additional information, included in the rightmost column of Table A-3 for each GL programming statement.

Code	Description
a	attribute (affected by pushattributes and popattributes)
g	only takes affect when gconfig() is executed
v	can be changed between bgn*/end* calls
w	applies to next winopen/swinopen/winconstraints call

Table A-2. Additional information codes

Table A-3 lists all the GL programming statements with their respective state types and other information included for reference. Table A-4 lists the GL programming statements you can call before `winopen`, `ginit`, or `gbegin`.

State Code	Statement	Other
w	acbuf	
f	acsize	
w	afunction	
p	addtopup	
r	arc*	
d	attachcursor	
f	backbuffer	a
w	backface	
w	bbox2*	
r	bgnclosedline	
r	bgnline	
r	bgnpoint	
r	bgnpolygon	
r	bgnqstrip	
r	bgnsurface	
r	bgntmesh	
r	bgntrim	
s	blankscreen	
s	blanktime	
w	blendfunction	
c	blink	
p	blkqread	
f	c*	v,a
p	callfunc	
p	callobj	
r	charstr	
p	chunksize	
r	circ*	
r	clear	
w	clearhitcode	

Table A-3. GL Programming Statments

State Code	Statement	Other
w	clipplane	
d	clkoff	
d	clkon	
p	closeobj	
f	cmode	g,a
w	cmov*	
f	color*	v,a
p	compactify	
w	concave	
f	cpack	v,a
r	crv	
r	crvn	
p	curorigin	
w	cursoff	
w	curson	
p	curstype	
w	curvebasis	
r	curveit	
w	curveprecision	
c	cyclemap	
r	czclear	
d	dbtext	
p	defbasis	
p	defcursor	
p	deflinestyle	
p	defpattern	
p	defpup	
p	defrasterfont	
p	delobj	
p	deltag	

Table A-3. GL Programming Statments (continued)

State Code	Statement	Other
w	depthcue	
p	dglclose	
p	dglopen	
p	dopup	
f	doublebuffer	g
r	draw*	
w	drawmode	a
p	editobj	
r	endclosedline	
w	endfeedback	
r	endfullscrn	
r	endline	
w	endpick	
r	endpoint	
r	endpolygon	
o	endpupmode	
r	endqstrip	
w	endselect	
r	endsurface	
r	endtmesh	
r	endtrim	
w	feedback	
w	finish	
w	fogvertex	
w	font	a
p	foreground	w
p	freepup	
f	frontbuffer	a
w	frontface	

Table A-3. GL Programming Statments (continued)

State Code	Statement	Other
w	fudge	w
w	fullscrn	
s	gammaramp	
p	gbegin	
w	gconfig	
p	genobj	
p	gentag	
w	getbackface	
f	getbuffer	
d	getbutton	
w	getcmmode	
f	getcolor	a
w	getcpos	
w	getcursor	
w	getdcm	
o	getdepth	
w	getdescender	a
p	getdev	
f	getdisplaymode	a
w	getdrawmode	
w	getfont	a
s	getgdesc	
w	getgpos	
w	getheight	a
w	gethitcode	
w	getlsbackup	a
w	getlsrepeat	a
w	getlstyle	a
w	getlwidth	a
f	getmap	

Table A-3. GL Programming Statments (continued)

State Code	Statement	Other
m	getmatrix	
c	getmcolor	
w	getmmode	
s	getmonitor	
w	getnurbsproperty	
p	getopenobj	
w	getorigin	
o	getothermonitor	
w	getpattern	
w	getplanes	
o	getport	
w	getresetls	
w	getscrbox	
w	getscrmask	
o	getshade	
w	getsize	
w	getsm	a
d	getvaluator	
s	getvideo	
w	getviewport	
f	getwritemask	a
w	getwscrn	
f	getzbuffer	
p	gexit	
w	gflush	
p	ginit	
-	glcompat	
w	GLC_OLDPOLYGON	
p	GLC_ZRANGEMAP	

Table A-3. GL Programming Statments (continued)

State Code	Statement	Other
w	greset	
f	gRGBcolor	a
o	gRGBcursor	
f	gRGBmask	a
w	gselect	
w	gsync	
s	gversion	
w	iconsize	w
w	icontitle	
w	imakebackground	w
w	initnames	
o	ismex	
p	isobj	
p	isqueued	
p	istag	
w	keepaspect	w
w	lRGBrange	
d	lampoff	
d	lampon	
w	linesmooth	
w	linewidth	a
-	lmbind	
w	BACKMATERIAL	
w	MATERIAL	v
w	LMODEL	
w	LIGHT	
w	lmcolor	v

Table A-3. GL Programming Statments (continued)

State Code	Statement	Other
-	lmdef	
p	BACKMATERIAL	
p	MATERIAL	v
p	LMODEL	
p	LIGHT	
m	loadmatrix	
w	loadname	
w	logicop	
m	lookat	
w	lrectread	
w	lrectwrite	
w	lsbackup	
w	lsetdepth	
w	lshaderange	
w	lsrepeat	a
p	makeobj	
p	maketag	
c	mapcolor (there is a separate screen-wide color map for each framebuffer)	
w	mapw	
w	mapw2	
w	maxsize	w
w	minsize	w
w	mmode	
r	move*	
w	mswapbuffers	
f	multimap	g
m	multmatrix	
w	n*	v
p	newpup	
p	newtag	
w	nmode	

Table A-3. GL Programming Statments (continued)

State Code	Statement	Other
w	noborder	w
p	noise	
w	noport	w
o	normal	
r	nurbscurve	
r	nurbssurface	
p	objdelete	
p	objinsert	
p	objreplace	
f	onemap	g
m	ortho	
m	ortho2	
w	overlay	g
t	pagecolor	
w	passthrough	
r	patch	
w	patchbasis	
w	patchcurves	
w	patchprecision	
r	pclos	
r	pdr*	
m	perspective	
w	pick	
w	picksize	
w	pixmode	
r	pmv*	
r	pnt*	
w	pntsmooth	
m	polarview	

Table A-3. GL Programming Statments (continued)

State Code	Statement	Other
r	polf*	
r	poly*	
w	polymode	
w	polysmooth	
w	popattributes	
m	popmatrix	
w	popname	
w	popviewport	
w	prefposition	w
w	prefsize	w
o	pupmode	
w	pushattributes	
m	pushmatrix	
w	pushname	
w	pushviewport	
r	pwlcurve	
d	qcontrol	
p	qdevice	
p	qenter	
p	qgetfd	
p	qread	
p	qreset	
p	qtest	
r	rcrv*	
r	rdr*	
r	readRGB	
r	readpixels	
f	readsource	
r	rect*	
r	rectcopy	

Table A-3. GL Programming Statments (continued)

State Code	Statement	Other
r	rectread	
r	rectwrite	
r	rectzoom	
w	resetls	
w	reshapeviewport	
f	RGBcolor	v,a
o	RGBcursor	
f	RGBmode	g,a
o	RGBrange	
f	RGBwritemask	a
d	ringbell	
r	rmv*	
m	rot	
m	rotate	
r	rpatch	
r	rpdr*	
r	rpmv*	
r	sbox*	
m	scale	
f	sclear	
w	scrbox	
w	screenspace	
w	scrmask	
d	scrnattach	
p	scrnselect	
w	scrsubdivide	
d	setbell	
w	setcursor	
d	setdblights	
o	setdepth	

Table A-3. GL Programming Statments (continued)

State Code	Statement	Other
w	setlinestyle	a
f	setmap	
s	setmonitor	
w	setnurbsproperty	
w	setpattern	a
p	setpup	
o	setshade	
d	setvaluator	
s	setvideo	
w	shademodel	a
o	haderange	
f	singlebuffer	g
o	smoothline	
o	spclos	
r	splf*	
f	stencil	
f	stensize	
w	stepunit	w
w	strwidth	a
w	subpixel	
w	swapbuffers	
w	swapinterval	
w	swaptmesh	
w	swinopen	
f	swritemask	
w	tevbind	
p	tevdef	
w	texbind	
p	texdef2d	

Table A-3. GL Programming Statments (continued)

State Code	Statement	Other
w	texgen	
t	textcolor	
t	textinit	
t	textport	
w	t*	
p	tie	
t	tpoff	
t	tpon	
m	translate	
w	underlay	g
p	unqdevice	
r	v*	
s	videocmd	
w	viewport	
o	winattach	
w	winclose	
w	winconstraints	
w	windepth	
m	window	
w	winget	
w	winmove	
w	winopen	
w	winpop	
w	winposition	
w	winpush	
w	winset	
w	wintitle	
f	wmpack	a

Table A-3. GL Programming Statments (continued)

State Code	Statement	Other
r	writeRGB	
f	writemask	a
r	writepixels	
r	xfpt*	
f	zbuffer	
f	zclear	
f	zdraw	
f	zfunction	
f	zsource	
f	zwritemask	

Table A-3. GL Programming Statments (continued)

Table A-4 lists the GL programming statements you can call before calling the first `winopen`, `ginit`, or `gbegin`).

Statements
fudge
foreground
imakebackground
iconsize
keepaspect
maxsize
minsize
noborder
noport
prefposition
prefsize
stepunit
getgdesc
gversion
ismex
scrnselect

Table A-4. GL Programming Statements called before winopen, ginit, or gbegin

Index

A

B

C

D

E

F

G

N

O

P

Q

R

S

T

U

V

W

Z