Graphics Library Programming Guide Volume I

Document Number 007-1210-060

Contributors

Written by Patricia McLendon
Illustrated by Dan Young, Howard Look, and Patricia McLendon
Edited by Steven W. Hiatt
Production by Derrald Vogt
Engineering contributions by John Airey, Kurt Akeley, Dan Baum, Rosemary Chang, Howard Cheng, Tom Davis, Bob Drebin, Ben Garlick, Mark Grossman, Michael Jones, Seth Katz, Phil Karlton, George Kong, Erik Lindholm, Howard Look, Rob Mace, Martin McDonald, Jackie Neider, Mark Segal, Dave Spalding, Gary Tarolli, Vince Uttley, and Rolf Van Widenfelt.

Graphics Library Programming Guide
Document Number 007-1210-060

Silicon Graphics, Inc.
Mountain View, California

Contents

Figures

Tables

Introduction

The *Graphics Library Programming Guide* introduces the Silicon Graphics® IRIS® Graphics Library™ (GL) to graphics programmers and application developers. The IRIS GL is a library of subroutines for creating 2-D and 3-D color graphics and animation.

This guide covers the IRIS GL as it is implemented on Silicon Graphics workstations. Where it is necessary to differentiate among the various workstation models, the applicable product is identified by its model name.

This guide is written for C programmers. It assumes:

- You are comfortable writing programs in the C programming language.

  Programmers who use one of the other languages supported by the GL (C++, Ada®, Fortran, Pascal, or BASIC) can follow the context of the information presented in this guide and look at the on-line GL manual (man) pages to obtain the proper syntax for that language.
- You are familiar with the Silicon Graphics IRIX™ operating system and can create and edit files.

This guide 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. For an introduction to computer graphics concepts, see the "Suggestions for Further Reading" at the end of this introduction.

How to Use This Guide

This guide is organized so that programmers new to the GL can read through it, skipping over the more complex topics until later. All sections dealing with advanced topics begin with a paragraph that first alerts you to the specialized nature of the material, and then guides you to the next appropriate section where you can continue reading.

Sample programs are provided throughout this guide, and as C language source code in the */usr/people/4Dgifts* directory on your workstation, to demonstrate GL programming concepts.

Once you understand the sample programs, you can experiment with the GL functions to get the effects you want to use in your own programs. The sample programs are general enough that you should be able to adapt them for your own needs. In many cases, you may be able to change parameter values to achieve the effects you want. In other cases, you need to work the GL functions into the structure of your own programs.

All the sample programs in *4Dgifts* are complete programs, but much of the sample code in the text is only fragments of complete programs. In these cases, you need to construct the remaining framework to have the program compile and execute.

How to Use the Sample Programs

All the sample programs in this guide are available on-line in the directory:

/usr/people/4Dgifts/examples/glpg

Follow the instructions in the *README* file to set up your environment so you can compile and run these programs.

Sample programs are in the directory under */usr/people/4Dgifts/examples/glpg* that corresponds to the chapter number that contains the program. All the directories begin with the letters *ch*, followed by a two digit number:

ch<nn>

where *nn* represents the chapter number. Sample programs for advanced topics and certain workstation models are located separately.

Typographical Conventions

In this guide, special typefaces designate:

- *New words, ideas*, or *important information*
- References to “Section Titles” in this guide and in other *documents.*
- *Directory, file*, and *program* names
- IRIX *commands* and *system()* calls
- *Subroutine arguments*
- **`Information that you enter from the keyboard`**
- **`<key>`** that you press on the keyboard
- GL `subroutines()`
- GL `TOKENS`, also called `SYMBOLS`
- `Sample code`

What this Guide Contains

In Volume I of the *Graphics Library Programming Guide:*

- Chapter 1, “Graphics Development Environment,” describes the tools and the facilities available for developing graphics applications.
- Chapter 2, “Drawing,” introduces graphics fundamentals that you use throughout the graphics development process.
- Chapter 3, “Characters and Fonts,” describes how to create fonts and work with character strings.
- Chapter 4, “Display and Color Modes,” explains the operation of the color monitor and tells you how to use different methods for describing and working with color.
- Chapter 5, “User Input,” describes the GL facilities for programming a user interface for your application.
- Chapter 6, “Animation,” describes how to set graphics scenes in motion.
- Chapter 7, “Coordinate Systems,” describes the coordinate systems used in creating and displaying geometry.

- Chapter 8, “Hidden-Surface Removal,” describes techniques you can use to reduce drawing time by drawing only the items that are visible.
- Chapter 9, “Lighting,” tells you how to use lights and lighting effects.
- Chapter 10, “Frame Buffers and Drawing Modes,” explains the different “layers” of graphics memory and of the display screen.
- Chapter 11, “Pixels,” describes the methods used to access screen pixels.
- Chapter 12, “Picking and Selecting,” describes how to program your applications to let users select items on the screen.

In Volume II of the *Graphics Library Programming Guide:*

- Chapter 13, “Depth-Cueing and Atmospheric Effects,” describes how you can add realism to your scene by adding depth perception cues.
- Chapter 14, “Curves and Surfaces,” tells you how to use NURBS to draw curves and surfaces.
- Chapter 15, “Antialiasing,” describes how to compensate for the inherent limitations in the way graphics are displayed on a monitor.
- Chapter 16, “Graphical Objects,” tells you how to work with hierarchies and use display lists.
- Chapter 17, “Feedback,” tells you how to access hardware operational information during the drawing process.
- Chapter 18, “Textures,” tells you how to use texture to promote realism.
- Chapter 19, “Using the GL in a Networked Environment,” describes how to run GL programs over the network on a remote host and describes how non-graphics servers can access graphics tools.
- Appendix A, “Scope of GL Subroutines,” describes the operational modes of the subroutines and the resources they act upon.
- Appendix B, “Global State Attributes,” describes attributes of the graphics development environment and the GL subroutines.
- Appendix C, “Transformation Matrices,” lists the matrices used to calculate graphics operations.
- Appendix D, “Error Messages,” lists the GL error messages and suggests debugging procedures.
- Appendix E, “Using Graphics and Share Groups,” tells you how to use shared processes with graphics.

How to Use the On-line Manual Pages

GL man pages are located in Section 3G of the on-line manual pages. Read the on-line GL man pages for detailed information about syntax, machine capabilities and special features. In most cases, the man pages provide more in-depth descriptions of the individual subroutines than is presented in this guide.

Using Man Pages from the Toolchest

An easy way to access man pages is to select "Manual Pages" from the Tools section of the Toolchest on your workstation. This launches the X Window System™ utility *xman*, a browsing tool for locating and reading man pages. To learn how to use *xman*, use the left mouse button to pull down the Options menu, select "Help", and read the on-line instructions.

You can leave *xman* running while you use other workstation utilities and windows. Click on the small square in the upper right-hand corner to iconify the browser when you aren't using it. See the *IRIS Utilities Guide* for more information about the Toolchest, system utilities, and other workstation basics.

Using Man Pages from an IRIX Shell

To read a man page from an IRIX shell, use the *man* command followed by the name of the command you are interested in. For example, to read the man page on man itself, enter:

```
% man man
```

To get a keyword list of man pages, enter **apropos** or **man -k** followed by the topic name. For example, to get a list of man pages that have the word "printer" in them, enter:

```
% man -k printer
```

or enter:

```
% apropos printer
```

This returns a list of man pages that contain the keyword, along with the subject line of each man page. Look at the subjects to decide which man page to view.

Note: If you have never used *apropo*s or *man -k* before, you (or your system administrator) may have to make the *whatis* database on your system. See the *makewhatis* man page for information on how to use that command.

Suggestions for Further Reading

For a general introduction to computer graphics, see:

Foley, J.D., A. van Dam, S. Feiner, and J.D. Hughes, *Computer Graphics Principles and Practice*, Second Edition, Addison Wesley Publishing Company Inc., Menlo Park, 1990.

Newman, W., and R. Sproull, *Principles of Interactive Computer Graphics*, 2nd ed., McGraw-Hill, New York, 1979.

In the manual set shipped with your system, see:

Graphics Library Programming Tools and Techniques, Silicon Graphics P/N 007-1489-010, for information about tools and techniques that can assist you in developing and debugging your IRIS GL applications.

Chapter 1

Graphics Development Environment

This chapter provides a brief overview of the graphics development environment. The graphics development environment contains the tools and systems that you work with when you write GL programs.

The tools available in the graphics development environment include your workstation, the IRIX operating system, the IRIS Graphics Library, system libraries, include files, the X Window System™, and a programming language that provides the interface to the GL. This guide describes the ANSI C interface to the GL.

The IRIS workstation processes graphics operations through the Geometry Pipeline™. The Geometry Pipeline is like an assembly line that performs the specialized graphics tasks that create and display graphics.

The IRIX operating system provides commands for setting up and maintaining your system, creating and editing files, and using IRIX system calls in your GL applications.

1.1 Using the Graphics Library and System Libraries

The GL is a library of subroutines that you call from a program, written in C, or another language, to draw and animate 2-D and 3-D color graphics scenes. You build your application using GL commands within the framework of the programming language. The programming language provides the logical structure for your program, and GL commands provide the interface to the graphics software and hardware.

The GL is network-transparent, so you can send graphics information over the network to a remote host, send graphics information to multiple display screens, or share processing tasks with other systems. To use network-transparent features, you must link with the shared libraries, as described in Section 1.3.2, "Compiling C Programs." See Chapter 19, "Using the GL in a Networked Environment," for more information about network transparency. The X Window System, described in Section 1.2, "Using the X Window System," manages operations over the network.

1.1.1 System Libraries

System libraries allow you to use capabilities such as graphics, fonts, and math routines. Shared libraries provide the optimum use of system resources and the best portability and compatibility between IRIS platforms. Libraries such as the shared graphics library (*libgl_s.a*), the math library (*libm.a*), and the font library (*libfm.a*) are invoked when you link to them when compiling your program.

1.1.2 Include Files

Include files provide standard definitions that your program uses. The include file *gl/gl.h* provides the standard definitions for graphics. Include files such as *math.h*, *device.h*, and *stdio.h* provide definitions for system facilities that your program uses.

1.2 Using the X Window System

Your workstation uses the X Window System. The X Window System provides resource management and communication through a client/server system, known as the X client and X server, that you can read about in your X Window System documentation. A few of the facilities that it includes are a terminal emulator (*xterm*), a window manager (*4Dwm*), a manual page browser (*xman*), mail managing utilities, user customizing options, font utilities, demos, and toolkits for creating graphical user interfaces.

The client/server model allows for remote display of graphics output. The DISPLAY environment variable determines where output is to be displayed.

Most of the time you display graphics on the default display, which is the screen attached to your workstation.

The X Window System designates displays by:

```
hostname:server.screen
```

where:

hostname	is the name of the server on which the display resides.
server	is the number of the X server. Some setups have more than one server, so they are numbered starting with 0; the default is 0.
screen	is the screen to display on. Some systems have more than one screen; the default is 0.

A typical display is:

```
mysystem.mydomain:0.0
```

If you are displaying on the system in front of you, it is called a *local host*. When displaying on a local host, you do not have to specify the *hostname*, so your DISPLAY is `:0.0` if you have only one screen on your system. Because this is the default, your system already knows that DISPLAY is `:0.0`, and you do not have to set it explicitly. You can learn more about X servers and displays in your X Window System documentation. See Chapter 19, “Using the GL in a Networked Environment,” for information about how to share graphics across a network.

The *4Dwm* window manager supplied with your system provides a default appearance for windows and manages window operations. You can create X windows, GL windows, and *mixed-model applications*, which are either X windows that accept GL input, or GL windows that intermix X and GL subroutines. Creating, manipulating, and displaying windows are activities central to your task as a GL programmer. Working with GL windows is covered in depth in the *Graphics Library Windowing and Font Library Programming Guide*.

1.3 Programming in C

The C programming language provides the framework for developing GL programs. This guide assumes that you are comfortable writing C programs.

1.3.1 Using the ANSI C Standard

Ideally, GL programs are independent of the particular IRIS platform and the installed graphics hardware that is running the application. The best way to develop machine-independent code is to follow the ANSI C standard and to query the system about available hardware to establish its graphics performance capabilities. You can learn about the ANSI C standard in your C programming language documentation. To compile non-ANSI compliant code, use the **-cckr** flag to invoke the standard C compiler, as described in Section 1.3.2.

1.3.2 Compiling C Programs

To compile C programs, use the *cc* (C compiler) command:

```
% cc myprogram.c -lc_s -lgl_s -lm -o myprogram
```

The first two options in this list allow the same binary to run on all IRIS-4D Series systems:

-lc_s	links with the shared C library.
-lgl_s	links with the shared Graphics library.
-lm	links with the math library.
-o	defines the name of the output file.

Some other compile options that you might want to use are:

-cckr	invokes the standard C compiler rather than ANSI C.
-lfm_s	links with the IRIS shared Font Manager library.
-lsun	links with the Network Information Service (NIS) to supply a hostname list for network-transparent GL applications.

1.4 GL Program Structure

Steps that you perform in a GL program include:

1. Querying the system for the availability of graphics resources.
2. Initializing the Graphics Library.
3. Calling GL subroutines to set and get global state attributes and to create and render graphics.
4. Exiting the Graphics Library.

The best way to learn about what a GL program contains is to look at a sample program. The sample program for this chapter is in the *ch01* directory of */usr/people/4Dgifts/examples/glpg*. Change directories (*cd*) to that directory now so you can follow along with the discussion.

This program is named *green.c*:

```
#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
```

Run the sample program by typing:

```
% green
```

Move the mouse cursor to the location where you want the window to appear and click the left 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 initializes the GL. 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—that is, by calling `color()` with the color specification you wish to use. See Chapter 4, "Display and Color Modes," for more information. The call to `gexit()` closes the window and tells the system that the process is finished using graphics.

As you work through the rest of the chapters, run the sample programs, copy them, and modify their parameters to see what effects are possible. The next sections examine the basic elements of a GL program that are in the sample program.

1.4.1 Initializing the System

Initializing the Graphics Library means telling the system to set up the graphics software and hardware environment in which a program will run. Use `winopen()` to initialize the GL and open a graphics window to tell the system where to display the graphics output of your program.

`winopen()` initializes the hardware, allocates memory for symbol tables and display list objects, and sets up default values for global state attributes. `winopen()` must be called before you call most GL routines.

You can call these GL subroutines before you call `winopen()`, `ginit()`, or `gbegin`:

```
fudge
foreground
imakebackground
iconsize
keepaspect
maxsize
minsize
noborder
noport
prefposition
prefsize
stepunit
getgdesc
gversion
ismex
scrnselect
```

1.4.2 Getting Graphics Information

You can make your application more portable by including code that queries the system for its graphics capabilities before commencing with the program.

Querying the System for Graphics Resources

`getgdesc()` allows you to inquire about characteristics of the graphics system, such as screen size, number of bitplanes, and the existence of optional hardware such as *z*-buffer. `getgdesc()` returns a number that describes the hardware configuration specified by its parameter, *inquiry*. See the *getgdesc*(3G) man page for a complete list of parameters. `getgdesc()` returns a value of -1 if the *inquiry* is invalid, or if the specified hardware is not installed. You can call `getgdesc()` at any time, including before the first `winopen()`.

The values that `getgdesc()` returns are not affected by changes to software modes or software configuration.

Querying the System for Graphics Hardware and Software Versions

`gversion(`*`v`*`)` returns information about the current graphics hardware and the Graphics Library version.

The argument *v* is 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.

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

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

Table 1-1 lists the descriptors returned by `gversion()`. 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*
VGXT, Skywriter	GL4DVGXT-*m.n*
RealityEngine	GL4DRE-*m.n*
Personal IRIS	GL4DPI2-*m.n*
Personal IRIS with Turbo Graphics	GL4DPIT-*m.n*
Personal IRIS (early serial numbers)	GL4DPI-*m.n*
IRIS Indigo Entry	GLDLG-*m.n*
XS	GL4DXG-*m.n*
XS24	GL4DXG-*m.n*
Elan	GL4DXG-*m.n*

Table 1-1 System Types and Graphics Library Versions

Note: Personal IRIS units with early serial numbers do not support the complete Personal IRIS graphics functionality.

Setting Compatibility Modes

`glcompat()` gives control over details of the compatibility among systems. `glcompat()` controls two compatibility modes. The first, `GLC_OLDPOLYGON`, offers compatibility with old-style polygons, described in Chapter 2, "Drawing." The second, `GLC_ZRANGEMAP`, controls the state of *z*-range mapping mode, described in Chapter 8, "Hidden-Surface Removal."

Setting and Getting the Graphics Resource Configuration

The GL is a modal system—it maintains settings, or *modes*, that determine how graphics resources are set up. When you change certain modes, you need to call `gconfig()` to configure them. Table A-2 in Appendix A, "Scope of GL Subroutines," contains a notation in its third column that indicates which commands require a `gconfig()` in order to take effect. Call `gconfig()` following the group of all calls that require a `gconfig()`.

The current configuration can be read back with `getgconfig()`.

Table 1-2 lists the `getgconfig()` tokens and the resource they describe.

Token	**Resource**
`GC_BITS_CMODE`	Color index size
`GC_BITS_RED`	Red component size
`GC_BITS_GREEN`	Green component size
`GC_BITS_BLUE`	Blue component size
`GC_BITS_ALPHA`	Alpha component size
`GC_BITS_ZBUFFER`	z-buffer size
`GC_ZMIN`	Minimum depth value that can be stored in the z-buffer
`GC_ZMAX`	Maximum depth value that can be stored in the z-buffer
`GC_BITS_STENCIL`	Stencil size
`GC_BITS_ACBUF`	Accumulation buffer size
`GC_MS_SAMPLES`	Number of samples in multisample buffer

Table 1-2 Tokens for Graphics Resource Inquiries

Token	Resource
`GC_BITS_MS_ZBUFFER`	Multisample zbuffer size
`GC_MS_ZMIN`	Minimum depth value that can be stored in the multisample z-buffer
`GC_MS_ZMAX`	Maximum depth value that can be stored in the multisample z-buffer
`GC_BITS_MS_STENCIL`	Multisample stencil size
`GC_STEREO`	Stereoscopic buffer state
`GC_DOUBLE`	Display double buffer state

Table 1-2 (continued) Tokens for Graphics Resource Inquiries

Requests for `GGC_ZSIZE`, `GGC_ACSIZE`, and `GGC_MSZSIZE` return a negative size if the buffer is signed.

1.4.3 Global State Attributes

The *global state attributes* are options that specify information that the GL uses. Many of the GL subroutines allow you to change the values of these attributes. Unless you specify otherwise, the global state attributes use their default values. See Appendix B for the default values of the global state attributes.

1.4.4 Exiting the Graphics Environment

`gexit()` performs housekeeping functions associated with the termination of graphics programming, such as freeing memory used for GL data structures. Call `gexit()` as the last step in a GL program.

Chapter 2

Drawing

This chapter describes how to draw graphics *primitives*. Primitives are basic geometric elements such as *points, lines*, and *polygons*. You can draw primitives with different colors and techniques. You can draw points with different sizes, lines with different widths and styles, and polygons with different patterns and filling methods.

- Section 2.1, “Drawing with the GL,” describes how to draw geometric figures with GL subroutines.
- Section 2.2, “Old-Style Drawing,” describes GL subroutines that were used for drawing in early releases of the GL, and is included for compatibility only.

2.1 Drawing with the GL

When you provide the specifications of a geometric figure, also called a *geometry,* the GL draws it on the screen right away. This is called *immediate mode*—the GL is drawing things as the drawing subroutines are called.

You describe a geometry in GL terms by specifying its *edges* and *corners*. Each corner is a *vertex* (a point in space). You specify the coordinates (position in space) of the vertices and the order in which the vertices are connected to form *edges*. Edges then connect to form the geometry. Edges connected as lines make a *wireframe* geometry; edges connected as polygons make a geometry with solid faces.

When you draw a geometry, you use one of the GL primitives to draw and connect the vertices. You mark the beginning and end of the vertex list with special `bgn*` and `end*` subroutines, which signify not only the beginning and end of the list of vertices, but also the type of primitive, as indicated by the asterisk (*). Which `bgn*` and `end*` subroutines you use depends on what kind of geometry you are drawing. You will see the `bgn*/end*` programming structure throughout the GL. It is analogous to the begin-end paradigm of modular programming languages.

Note: You need to call `gflush()` after the `end*` statement to complete the drawing process. If you have used the GL before, you may remember that this step was previously necessary only when you were using the DGL. Because the GL is network-transparent, all programs need to call `gflush()` after the last drawing command.

You tell the GL to begin drawing with a `bgn`*`geometry`* statement, where *`geometry`* denotes the primitive to use. You then specify a list of vertices, whose coordinates are of the appropriate type of vertex data, to connect in order to form the edges. Finally, you tell the GL to close the figure with an `end`*`geometry`* statement.

2.1.1 Vertex Subroutines

Specify a vertex list by calling the `vertex()` subroutine for each vertex between the `bgn*` and `end*` statements.

The content of the `bgn*/end*` modules have the format illustrated by this pseudocode:

```
bgngeometry();
   vtype(vertex 1);
   vtype(vertex 2);
   vtype(vertex .);
   vtype(vertex .);
   vtype(vertex n);
endgeometry();
gflush();
```

Note: No drawing is guaranteed to happen until `gflush()` is called.

The GL contains 12 forms of the `vertex()` subroutine—one for each possible data type of a vertex coordinate. This group of subroutines is known collectively as the `v()`, for vertex, subroutine.

You can specify vertex coordinates as 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 `v()` subroutine for defining vertices in 2-D, 3-D, and 4-D, also called *homogeneous coordinates*.

Homogeneous coordinates are referred to as 4D because they use a fourth parameter in addition to the 3-D coordinates. Homogeneous coordinates are useful for matrix manipulations and other operations common to graphics.

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 indicates the data type: *s* for short integer (16 bits), `i` for long integer (32 bits), `f` for single-precision floating point (32 bits), `d` for double-precision floating point (64 bits).

Table 2-1 lists the vertex subroutines.

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 Vertex Subroutines

This sample program, *greensquare2.c*, demonstrates the use of some of the different vertex subroutines. This program draws a square with green lines.

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

short vert1[3] = {200, 200, 0};      /* lower left corner */
long vert2[2] = {200, 400};          /* upper left corner */
float vert3[2] = {400.0, 400.0};     /* upper right corner */
double vert4[3] = {400.0, 200.0, 0.0}; /* lower right corner */

main()
{
   prefsize(400, 400);
   winopen("greensquare2");
   ortho2(100.5, 500.5, 100.5, 500.5);
   color(WHITE);
   clear();
   color(GREEN);
   bgnline();
      v3s(vert1);
      v2i(vert2);
      v2f(vert3);
      v3d(vert4);
      v3s(vert1);
   endline();
   sleep(10);
   gexit();
   return 0;
}
```

You have seen the `prefsize()` and `winopen()` commands before. The `ortho2()` command sets up a coordinate system inside your window to allow you to see the output of the program at a reasonable size and perspective. See Chapter 7, "Coordinate Transformations," for more information on `ortho2()`. The values have .5 added to them so that pixels are centered on whole numbers. This eliminates the potential problem of *roundoff error* from non-integer pixel locations causing the display to be shifted by one or more pixel.

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

- Within one geometric figure, you can mix different types of vertices. In a typical application, all the vertices tend to have the same dimension and the same form.
- In the GL, all geometric figures are 3-D and the hardware treats them as such. 2-D versions of the vertex subroutines are actually shorthand for an equivalent 3-D subroutine with the *z* coordinate set to zero.

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

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

long vert1[2] = {101, 101};      /* lower left corner */
long vert2[2] = {101, 500};      /* upper left corner */
long vert3[2] = {500, 500};      /* upper right corner */
long vert4[2] = {500, 101};      /* lower right corner */

main()
{
   prefsize(400, 400);
   winopen("crisscross");
   ortho2(100.5, 500.5, 100.5, 500.5);
   color(WHITE);
   clear();
   color(RED);
   bgnline();
      v2i(vert1);
      v2i(vert3);
   endline();
   bgnline();
      v2i(vert2);
      v2i(vert4);
   endline();
   sleep(10);
   gexit();
   return 0;
}
```

This example declares four long arrays *(vert1*, *vert2*, *vert3*, and *vert4*) and assigns values to all the elements of each array. The size of the next window to be opened is established by `prefsize()` as 400 pixels by 400 pixels. Next, `winopen()` opens a window with the title *crisscross*.

The `ortho2()` call sets up the 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. The `color()` call sets the current color to white and paints the window with the current color, which is white.

The next four lines of code draw a line from (101, 101) to (500, 500)—the lower-left corner to the upper-right corner. `bgnline()` tells the system to prepare to draw a line using the list of vertices immediately following it. `v2i()` takes an array of coordinates as its argument and creates vertices at the specified coordinates.

The first `v2i()` subroutine call creates the first endpoint of the line segment. The second `v2i()` subroutine call creates the other 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 (101, 500) to (500, 101)—the lower-right corner to the upper-left corner. The IRIX call *sleep(10)* delays the program from exiting until 10 seconds have elapsed.

You can use any of the five primitives described in the next five sections—*points, lines, closed lines, polygons,* and *meshes*—with the vertex subroutines.

2.1.2 Points

Use `bgnpoint()` and `endpoint()` to specify a vertex list that is drawn as a group of disconnected points.

This sample program, *pointpatch.c*, draws a set of 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++) {
            vert[1] = 100 + 10 * j;/* load the y coordinate */
            v2i(vert);             /* draw the point */
        }
        endpoint();
    }
    sleep(10);
    gexit();
    return 0;
}
```

Mathematical points have no size, but a point must be assigned a size to be displayed. The system draws a point as a 1-pixel point on the screen. On some systems, you can define the size of the points that are drawn using the `pntsize()` or `pntsizef()` subroutines. Specify the size of the point in pixels as a short for `pntsize()` and as a float for `pntsizef()`. See the *pntsize*(3G) and *pntsizef*(3G) man pages for information about point size limits and the capabilities of different systems.

2.1.3 Lines

The GL has two types of line primitives: *polylines* and *closed lines*. A polyline is a series of connected line segments. A closed line automatically connects the last vertex in a polyline to the first vertex in the polyline.

Polylines

Use `bgnline()` and `endline()` to specify a vertex list that is drawn as a polyline. Line segments can cross each other, and vertices can be reused. If the vertices are specified with 3-D or 4D coordinates, you can place them anywhere in 3-D space; they need not all lie in the same plane.

This sample program, *greensquare.c*, draws lines that form a green square. The `bgnline()/endline()` pair is used to select the polyline primitive for drawing. The first vertex, `v2i(vert1)`, is repeated at the end of the vertex list to connect the first and last line segments. It is better to draw such a sequence using the `closedline()` primitive, as described next in “Closed Lines”.

```
#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()
{
    prefsize(400, 400);
    winopen("greensquare");
    ortho2(100.5, 500.5, 100.5, 500.5);
    color(WHITE);
    clear();
    color(GREEN);
    bgnline();
        v2i(vert1);
        v2i(vert2);
        v2i(vert3);
        v2i(vert4);
        v2i(vert1);
    endline();
    sleep(10);
    gexit();
    return 0;
}
```

Closed Lines

Use `bgnclosedline()` and `endclosedline()` to specify a vertex list that is drawn as a series of line segments, in which the last vertex in the sequence is automatically connected to the first vertex.

This sample program, *n-gon.c*, draws a regular *n*-gon, an *n*-sided polygon with sides of equal length:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <gl/gl.h>
#define X    0
#define Y    1
#define XY   2

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

/* Tell user to enter number of sides if no number is typed */
    if (argc != 2) {
        fprintf(stderr, "Usage: n-gon <number of sides>\n");
        return 1;
    }
    n = atoi(argv[1]); /* Convert character entered to integer */
    prefsize(400, 400);
    winopen("n-gon");
    color(WHITE);
    clear();
    color(RED);
    bgnclosedline();
    for (i = 0; i < n; i++) {
        vert[X] = 200.0 + 100.0 * fcos(i * 2.0 * M_PI / n);
        vert[Y] = 200.0 + 100.0 * fsin(i * 2.0 * M_PI / n);
        v2f(vert);
    }
    endclosedline();
    sleep(10);
    gexit();
    return 0;
}
```

Run the program by typing **ngon *n***, where *n* is the number of sides you want in the *n*-gon. If you do not specify a number, the program exits and displays a usage message telling you how to run it.

This example draws the *n*-gon only once, so there is no real penalty for computing the coordinates of the vertices. If it were necessary to draw the polygon over and over again, the calculated vertices should probably be saved in an array. In applications that draw the same geometry repeatedly with different viewing parameters, it is usually more efficient to save the coordinates in arrays. The sample program does not save the coordinates because the *n*-gon is drawn only once.

Linestyles

A *linestyle* describes the way the system draws lines on the screen. The linestyle represents a 16-bit pattern on the screen. The least significant bit of the pattern is the *mask* for the first pixel of the line and every sixteenth pixel thereafter.

The mask determines which pixels are turned on and which pixels are left off. Pixels corresponding to 1 in the linestyle are drawn; those corresponding to 0 are not drawn. For example, the linestyle 0xFFFF draws a solid line; 0xF0F0 draws a dashed line; and 0x8888 draws a dotted line. The system runs this pattern repeatedly to determine which pixels in a 16-pixel line segment to color. There is no concept of an opaque linestyle in the GL.

Defining a Linestyle

Use the `deflinestyle()` subroutine to define a line style to save in an indexed table. When you want that style to be used, you retrieve it by its index. There are 2^{16} possible linestyle patterns. By default, index 0 contains linestyle 0xFFFF, which draws solid lines. You cannot redefine linestyle 0.

To replace a linestyle in the table, specify the index of the new linestyle in place of the old one. Use `getlstyle()` to query the index of the current linestyle.

The system uses the current linestyle to draw lines and to outline rectangles, polygons, circles, and arcs. Linestyle 0 (solid line) is the default linestyle. Use `setlinestyle()` to select another linestyle. Its argument is an index into the linestyle table built by calls to `deflinestyle()`.

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()`.

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×0, and each 1 becomes a series of factor×1. For example, if the pattern is 0xFE00 and factor=1, the linestyle is 9 bits off followed by 7 bits on. If factor=3, the linestyle is 27 bits off followed by 21 bits on.

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

Use `linewidth()` or `linewidthf()` to specify the width of a line. Specify the width of the line in pixels as a short for `linewidth()`, and as a float for `linewidthf()`. 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.

Use `getlwidth()` to query the current linewidth in pixels.

The ANSI C specifications for the line style and line width subroutines are:

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

long getlsrepeat(void);

long getlstyle(void);

long getlwidth(void);

void linewidth(short n);

void linewidthf(float n);
```

2.1.4 Polygons

A *polygon* is specified by a connected sequence of vertices that all lie in a plane. You define the boundary of the polygon by connecting the vertices in order: *v1* to *v2*, *v2* to *v3*, and so on, finally connecting *vn* back to *v1*. These connecting segments are called *edges*. The interior of a polygon is the area inside its edges. The GL lets you specify up to 255 vertices per polygon.

Note: Even though the GL supports polygons with up to 255 vertices, performance is usually optimized only for polygons of 3 or 4 vertices. Polygons with more than 4 vertices are typically better represented as *meshes*. See Section 2.1.6, "Meshes," for the definition of meshes and for information about how they are used.

Figure 2-1 through Figure 2-6 contain some examples of polygons. The heavy black dots represent vertices and the lines represent edges.

A polygon is *simple* if its edges intersect only at their common vertices. In other words, the edges cannot cross or touch each other. A polygon is *convex* if a line segment joining any two points in its interior is completely contained within the polygon.

Figure 2-1 is a convex and simple polygon.

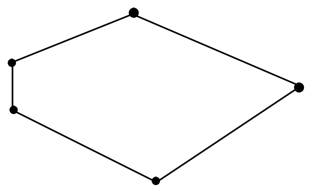


Figure 2-1 Simple Convex Polygon

Figure 2-2 and Figure 2-3 are both simple, but not convex, polygons. They are not convex because you can draw a line connecting two interior points (shown as a dashed line) that appears outside of the polygon.

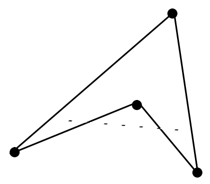


Figure 2-2 Simple Concave Polygon

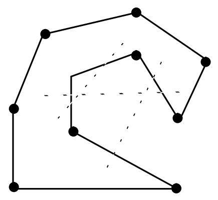


Figure 2-3 Another Simple Concave Polygon

Non-convex polygons are also called *concave*. Algorithms that render only convex polygons are much simpler than those that can render both convex and concave polygons.

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()`.

Note: 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 performance penalty for setting the concave flag, but it makes the code portable to other Silicon Graphics machines. If you do not want to pay the performance penalty of using `concave()` on any machine, break up the concave polygons into smaller, convex polygons yourself.

The GL and the Silicon Graphics hardware can correctly render any polygon if it is simple, or if it consists of exactly four points.

Figure 2-4, Figure 2-5, and Figure 2-6 are not simple polygons.

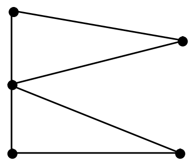


Figure 2-4 Nonsimple Polygon

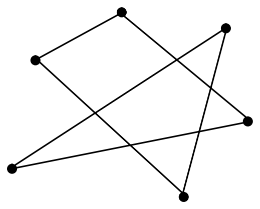


Figure 2-5 Another Nonsimple Polygon

Figure 2-6 shows a special type of polygon called a *bowtie, a* 4-vertex nonsimple polygon.

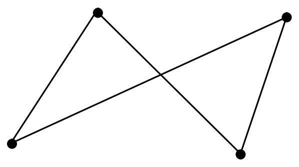


Figure 2-6 Bowtie Polygon

The GL can render the simple polygons in Figure 2-1, Figure 2-2, Figure 2-3, and the bowtie polygon in Figure 2-6. Bowtie polygons are handled in a hardware-specific manner. The polygon appears either as a bowtie, or as a quadrilateral with a segment missing from one edge. The results of rendering the type of non-simple polygons shown in Figure 2-4 and Figure 2-5 are unpredictable.

Draw polygons using the same basic syntax as the other primitives, that is, specify a list of vertex subroutines between a `bgnpolygon()/endpolygon()` pair. The system draws a polygon as a filled area on the screen. As it does with closed lines, the GL automatically connects the first and the last point. You do not need to repeat the first point at the end of the sequence.

This sample program, *bluehex.c*, 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()
{
    int i;

    prefsize(400, 400);
    winopen("bluehex");
    ortho2(0.0, 60.0, 0.0, 60.0);
    color(BLACK);
    clear();
    color(BLUE);
    bgnpolygon();
    for (i = 0; i < 6; i++)
        v2f(hexdata[i]);
    endpolygon();
    sleep(10);
    gexit();
    return 0;
}
```

If you approximate a surface with 4-sided polygons, some of them may not lie in a plane. If the vertices of a polygon do not lie in a plane, it is likely that in certain orientations the polygon on the screen may look like its edges cross. This is especially true at the *silhouette edges* of a mesh, where the mesh wraps around to the back of the shape. However, with 4 vertices, these distorted polygons will all be interpreted as bowtie polygons that are correctly rendered by the GL.

The GL can render the bowties that arise from surface-approximating meshes. In most other circumstances, however, the GL routines generate only true (planar) polygons.

If a polygon lies in a plane, the only way distortion can occur is from floating point inaccuracies.

IRIS-4D Series systems convert all arguments to 32-bit floating point for hardware calculations. Consequently, only long integers in the range of -2^{23} and $2^{23}-1$ retain full precision after conversion. Integers outside this range retain 24 bits of precision after conversion.

Patterns

You can fill polygons (as well as rectangles and arcs) with *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.

Use `defpattern()` to define a pattern to be saved in an indexed table. Specify an index into a table of patterns (*n*), the length of the array (*size*), and an array of short integers (*mask*). The *size* argument selects either a 16×16 (*size*=32) or a 32×32 (*size*=64) pattern.

The origin of the pattern is the lower-left corner of the screen. Define the bottom row of the pattern first. Specify each row of a 16×16 pattern with a single short. Specify each row of a 32×32 pattern with two shorts—first the left 16 bits, then the right 16 bits. Bit 0 of each short is the right-most bit of its respective position in the row. There is no concept of an opaque pattern in the GL.

Pattern 0 is the default pattern, which is solid. You cannot redefine the pattern at index 0.

Use `setpattern()` to select which defined pattern the system uses. The argument for `setpattern()` is the index you defined with `defpattern()`. Use `getpattern()` to query the index of the current pattern.

The ANSI C specifications for the pattern subroutines are:

```
void defpattern(short n, short size, unsigned short mask[]);
```

```
long getpattern(void);
```

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

2.1.5 Point-Sampled Polygons

This section tells you exactly which pixels are turned on when the system displays a polygon. It is important to know which pixels are turned on for display accuracy and drawing performance reasons, but you may want to skip this section on your first reading, because it contains advanced concepts.

To represent a polygon on the screen, the system must turn on a group 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.

To illustrate the problem, consider drawing the two rectangles shown in Figure 2-7. The numbered grid represents pixels and the black dots represent pixels that are on.

The rectangle on the left has sides of ($2 \le x \le 5$ and $1 \le y \le 4$).
The rectangle on the right has sides of ($2 \le x \le 5$ and $4 \le y \le 6$).

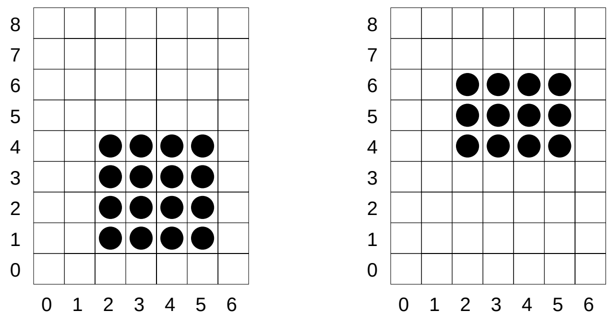


Figure 2-7 Non–Point-Sampled Polygons

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 both rectangles the way Figure 2-7 shows, the pixels on the line $y = 4$ are drawn twice—once for each polygon.

Drawing every overlapping edge twice is not a very efficient way to draw a large number of polygons. Drawing the polygons in this manner also gives the effect of outlining the polygon, which may not be the way you want the polygons to look. For example, if the polygons represent a transparent surface, the duplicated edge is twice as dense as the interior of the polygon, giving the entire surface an outlined appearance.

Even if the surface is not transparent, filling the polygons in this way can still create undesirable visual effects. If you draw a checkerboard pattern with edges that overlap by exactly one pixel, 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 both images are identical. See Chapter 6, "Animation," for more information about single buffer mode.

The GL resolves these problems by using *point-sampled polygons*. Point-sampled polygons assume that *ideal mathematical lines* (lines with no thickness) connect the vertices. The system draws any pixel whose center lies inside the mathematically precise polygon. Pixels whose centers lie outside the polygon do not get drawn. Pixels whose centers lie exactly on a mathematical line segment or vertex are filled 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 of a screen-aligned rectangle that is drawn on the exact pixel centers are filled, whereas the pixels on the right and top of such a rectangle are not filled.

Figure 2-8 shows point-sampled versions of the two rectangles in Figure 2-7. Point-sampling effectively eliminates the duplication of pixels from the right and top edges of the polygon, but adjacent polygons can fill those pixels.

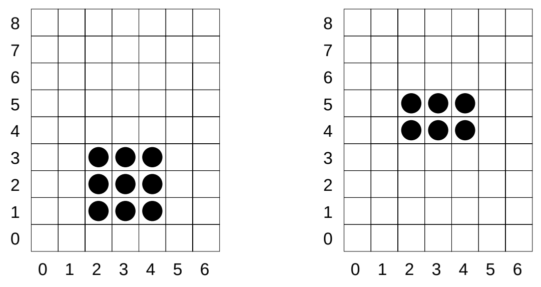


Figure 2-8 Point-Sampled Polygons

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 non-rectangular polygons, the drawn area of the polygon cannot be exact, but the drawn area of the point-sampled polygon is closer to the true area of the polygon than is the area of an outlined polygon. Outlined polygons that have increased area are sometimes called *fat polygons*.

Figure 2-9 illustrates the pixels that are turned on in a point-sampled representation of the polygon that connects the vertices (1,1) (1,4) (5,6) and (5,1). 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.

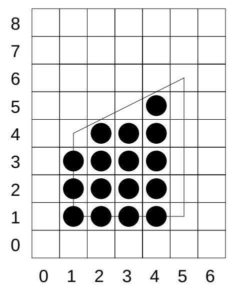


Figure 2-9 Point-Sampled Polygon with Edges Exactly on Pixel Centers

As mathematical entities, lines have no thickness. However, to represent a line on the screen, the system assumes a thickness of exactly one pixel (or whatever line width you have assigned). 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 you draw a line around a point-sampled polygon, it fills in the pixels at the upper and right-hand edges.

The default for the older subroutines such as `polf()`, `rect()`, `circle()`, is to draw a line around the point-sampled polygon. However, if you use the old-style subroutines, it is recommended that you use `glcompat()` as described in Section 2.2.2, “Nonrecommended Old-Style Subroutines,” to specify point-sampled polygons. You don’t have to do this for the vertex subroutines because they draw point-sampled polygons by default.

Anomalies can occur in the display of very thin filled polygons. Figure 2-10 shows a thin point-sampled triangle connecting the points (1,1), (2,3), and (12,7). The polygon looks like it has holes, but if you draw adjacent polygons that share the same vertices, all the pixels are eventually filled.

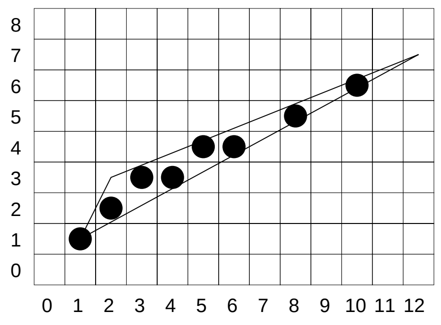


Figure 2-10 Point Sampling Anomaly

2.1.6 Meshes

This section covers an advanced topic. You may want to skip it on the first reading, because it mentions topics that are not fully covered until later in this guide.

This section describes how to draw geometric figures that are constructed entirely of adjacent 3-sided (triangular) or 4-sided (quadrilateral) polygons. When you draw connecting polygons, a *mesh* is formed. A mesh made of triangles is called a *triangular mesh (t-mesh)*, and a mesh made of quadrilaterals is called a *quadrilateral strip (q-strip)*.

Figure 2-11 shows an example of a simple t-mesh.

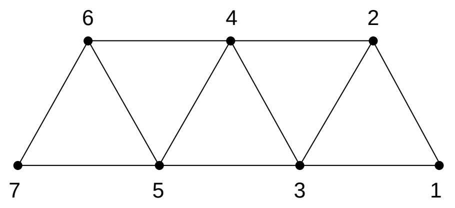


Figure 2-11 Simple Triangle Mesh

In this example, the seven vertices form five triangles (123, 324, 345, 546, 567). Vertex 1 and vertex 7 appear in one triangle, vertices 2 and 6 appear in two triangles, and all the rest of the vertices each appear in three different triangles. In a larger mesh, a higher percentage of the points would be used three times. Drawing the geometry in Figure 2-11 as a t-mesh is more efficient than drawing it as five separate triangles, because the t-mesh draws the shared vertices only once.

To draw a t-mesh, you specify a sequence of vertices between a `bgntmesh()/endtmesh()` pair.

The `bgntmesh()` call signifies that the vertices following it are connected as triangles until an `endtmesh()` call is received. The system uses two memory locations to remember the last two vertices and a pointer to keep track of what it is doing. The pointer alternates from one retained vertex to the other while it is drawing the mesh.

Refer to Table 2-2 and the discussion following it to see the sequence of events that happens internally when a t-mesh is drawn.

Sequence	Vertex	P→	R1	R2	Next Vertex
1	`bgntmesh()`	R1	~~~	~~~	v1
2	v1	R2	v1	~~~	v2
3	v2	R1	v1	v2	v3
4	v3	R2	v3	v2	v4
5	v4	R1	v3	v4	v5
6	v5	R2	v5	v4	v6
7	v6	R1	v5	v6	v7

Table 2-2 Sequence of Vertices in a Mesh

Let P→ represent the pointer and let R1 and R2 represent the two vertices that are retained:

1. The `bgntmesh()` call initializes the pointer to R1.
2. The first vertex (v1) is stored in the location pointed to by P, which is R1. The pointer is now changed to point to R2.
3. The next vertex (v2) is stored in the location pointed to by P, which is R2. The pointer is changed to point to R1.
4. The next vertex (v3) is the third vertex, so a triangle is drawn. Triangles are drawn R1, R2, next vertex, so triangle 123 is drawn. v3 is stored in the location pointed to by P, which is R1. The pointer is changed to point to R2.
5. The next vertex (v4) is now connected to R1 and R2. R1 has v3 stored in it, and R2 has v2 stored in it, so the triangle drawn (R1,R2,new) is triangle 324. v4 stored in the location pointed to by P, which is R2. The pointer is changed to point to R1.
6. This process continues until the `endtmesh()` call is encountered.

This process draws the triangles 123, 324, 345, 546, and 567 for the mesh in Figure 2-11.

Figure 2-12 illustrates a more complex situation. The first six triangles (123, 324, 345, 546, 567, 768) 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.

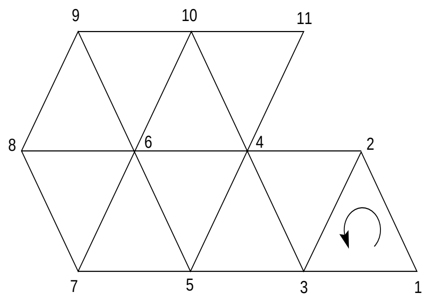


Figure 2-12 Example of `swaptmesh()` Construction

To draw meshes like the one in Figure 2-12, you must exchange the order of the two stored vertices. Use the `swaptmesh()` subroutine to exchange the pointer to the other retained vertex, as shown in the following code fragment:

```
bgntmesh();
   v3f(vert1);
   v3f(vert2);
   v3f(vert3);
   v3f(vert4);
   v3f(vert5);
   v3f(vert6);
   v3f(vert7);
swaptmesh();
   v3f(vert8);
swaptmesh();
   v3f(vert9);
swaptmesh();
   v3f(vert10);
   v3f(vert4);
   v3f(vert11);
endtmesh();
```

Figure 2-13 shows another example of how to use `swaptmesh()`.

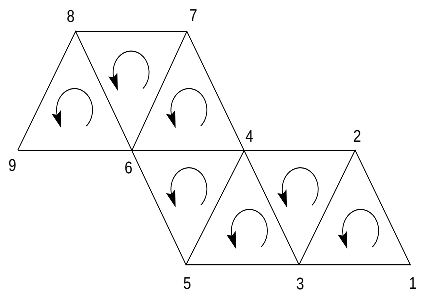


Figure 2-13 Another `swaptmesh()` Example

This sequence draws the mesh in Figure 2-13:

```
bgntmesh();
    v3f(vert1);
    v3f(vert2);
    v3f(vert3);
    v3f(vert4);
    v3f(vert5);
swaptmesh();
    v3f(vert6);
    v3f(vert7);
swaptmesh();
    v3f(vert8);
    v3f(vert9);
endtmesh();
```

The arrows show that all the faces in this t-mesh specify their vertices in counter-clockwise order. This is important if you want to do *hidden-surface removal*, described in Chapter 8, or *two-sided lighting,* described in Chapter 9, later. Because the faces are counter-clockwise, they specify *front-facing* polygons, which display when backface removal is turned on. For lighting, the counter-clockwise faces specify normals that follow the right-hand rule—that is, point out of the paper toward you. If you take your right hand and curl your fingers in the direction pointed to by the arrows, with your thumb sticking out, your thumb points straight out, away from the paper. Your thumb represents the vertex normals for the counter-clockwise faces.

This sample program, *octahedron.c*, draws a three-dimensional octahedron (8-sided regular polyhedron) using the t-mesh primitive. Because meshes in two dimensions are of little use, the example is three-dimensional. Knowing how to create a three-dimensional mesh is quite useful. This program also uses a number of routines that are covered in later chapters, including three-dimensional rotations, hidden surface removal, smooth (double-buffered) motion, and `cpack()`, so ignore the subroutines that do not apply to the specification of the geometry if you are studying the program to understand the logic of mesh drawing. The calculations of rotation angles simply cause the octahedron to tumble in an interesting way.

```
#include <stdio.h>
#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}
};
unsigned long octcolor[6] = {
   0xff0000,                    /* [0] = blue */
   0x00ff00,                    /* [1] = green */
   0x0000ff,                    /* [2] = red */
   0xff00ff,                    /* [3] = magenta */
   0xffff00,                    /* [4] = cyan */
   0xffffff,                    /* [5] = white */
};

void vertex(i)
int i;
{
   cpack(octcolor[i]);
   v3f(octdata[i]);
}

void drawoctahedron()
{
   bgntmesh();
      shademodel(GOURAUD);
      vertex(0);
      vertex(1);
   swaptmesh();
      vertex(2);
```

```
    swaptmesh();
       vertex(4);
    swaptmesh();
       vertex(5);
    swaptmesh();
       vertex(1);
       vertex(3);
    shademodel(FLAT);
       vertex(2);
    swaptmesh();
       vertex(4);
    swaptmesh();
       vertex(5);
    swaptmesh();
       vertex(1);
    endtmesh();
}
main()
{
    Angle xang, yang, zang;
    long zval;
    int cnt;
    if (getgdesc(GD_BITS_NORM_DBL_RED) == 0) {
       fprintf(stderr, "Double buffered RGB not available\n");
       return 1;
    }
    if (getgdesc(GD_BITS_NORM_ZBUFFER) == 0) {
       fprintf(stderr, "Z-buffer not available\n");
       return 1;
    }
    prefsize(400, 400);
    winopen("octahedron");
    doublebuffer();
    RGBmode();
    gconfig();
    ortho(-2.0, 2.0, -2.0, 2.0, -2.0, 2.0);
    zbuffer(TRUE);   /* hidden surfaces removed with z-buffer */
    zval = getgdesc(GD_ZMAX);
    xang = yang = zang = 0;
    for (cnt = 0; cnt < 1000; cnt++) {
       czclear(0x000000, zval);
       pushmatrix();/* save viewing transformation */
           rotate(xang, 'x'); /* rotate by xang about x axis */
           rotate(yang, 'y'); /* rotate by yang about y axis */
           rotate(zang, 'z'); /* rotate by zang about z axis */
           drawoctahedron();
```

```
        popmatrix();        /* restore viewing transformation */
        swapbuffers();             /* show completed drawing */

        xang += 10;
        yang += 13;
        if (xang + yang > 3000)
            zang += 17;
        if (xang > 3600)
            xang -= 3600;
        if (yang > 3600)
            yang -= 3600;
        if (zang > 3600)
            zang -= 3600;
    }
    gexit();
    return 0;
}
```

Quadrilateral Strips

In addition to the t-mesh, the GL supports quadrilateral strips (q-strips). Q-strips are similar in many ways to t-meshes, but might be better suited for the representation of shapes that are fundamentally quadrilateral rather than triangular in nature.

Use `bgnqstrip()` and `endqstrip()` to specify vertex list that forms quadrilateral strips.

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

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

For example, this sequence draws the three quadrilaterals: (1,2,4,3), (3,4,6,5), and (5,6,8,7) in Figure 2-14:

```
bgnqstrip();
   v3f(vert1);
   v3f(vert2);
   v3f(vert3);
   v3f(vert4);
   v3f(vert5);
   v3f(vert6);
   v3f(vert7);
   v3f(vert8);
endqstrip();
```

```
8--6--4--2
|  |  |  |
|  |  |  |
7--5--3--1
```

Figure 2-14 Mesh of Quadrilateral Strips

Note: The quadrilaterals are drawn as though each quadrilateral were an independent polygon with vertex order (*n*, *n*+1, *n*+3, *n*+2).

Note that the vertex order required by q-strips matches the order required for "equivalent" triangle meshes. The example vertex sequence that produces the mesh in Figure 2-14 produces triangles (123, 324, 345, 546, 567, 768), as shown in Figure 2-15 when bounded by `bgntmesh()` and `endtmesh()` calls.

```
8--6--4--2
| /| /| /|
|/ |/ |/ |
7--5--3--1
```

Figure 2-15 Equivalent T-mesh

In general, quadrilateral data looks better when drawn with q-strips than with a t-mesh. This is because Gouraud shading calculations operate on the original quadrilateral data, rather than on the decomposed triangles.

Note: IRIS-4D VGX, VGXT, SkyWriter, and RealityEngine graphics use 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.

2.1.7 Controlling Polygon Rendering

Use the `polymode()` subroutine to 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).

The ANSI C specification for `polymode()` is:

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

where *mode* is defined by 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. Therefore the antialiasing mode, `pntsmooth()`, or `linesmooth()` as well as linewidth and linestipple are significant. `PYM_FILL` is the standard fill operation that was previously the only option. See Chapter 15, "Antialiasing," for information on antialiasing.

Figure 2-16 shows how `PYM_LINE` affects clipping. 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 clipping planes, which are usually the borders of the viewport.

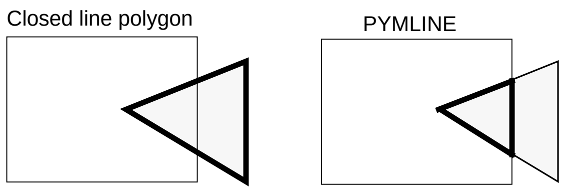


Figure 2-16 Clipping Behavior of `PYM_LINE`

`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 (that is, the outline does not extend beyond the exact polygon boundaries).
- Changed pixels 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 accurately composed of filled polygons. Both hidden-line and scribed-surface renderings can be done taking advantage of this fact. See Chapter 8, "Hidden-Surface Removal," for more information on hidden surfaces.

Note: Not all systems 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`. See the *polymode*(3G) man page for details.

2.2 Old-Style Drawing

This section describes drawing methods that were used in previous releases of the GL. Skip this section if you are developing new GL applications.

The vertex drawing subroutines described in the last section are the preferred way to draw any geometry except curves and surfaces, which are covered in Chapter 14, "Curves and Surfaces." All new development should use the vertex method. It is OK to use the subroutines described under Section 2.2.1, "Basic Shapes," but the subroutines described in Section 2.2.2, "Nonrecommended Old-Style Subroutines," are not recommended.

The architecture of earlier Silicon Graphics 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 GL.

In most cases, the internals of these earlier subroutines have been rewritten to use the vertex subroutines. However, to guarantee that you get the optimal performance of the new programs, use the vertex 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 vertex subroutines. 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, and the second is the value to which it is set.

The default `GLC_OLDPOLYGON` value is 1, in which outlining is turned on. To turn off polygon outlining, use:

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

Performance for the basic shapes subroutines can also be improved significantly by calling `glcompat(GLC_OLDPOLYGON, 0)` to draw point-sampled polygons and defeat polygon outlining.

2.2.1 Basic Shapes

In this section and the next, the subroutine names follow a pattern. The root name of the subroutine indicates the shape that is drawn. Letters appended to the root name indicate whether the shape is filled or drawn as an outline and also indicate its data type. The default data type is floating point (32 bits):

f	indicates that the figure is filled rather than drawn as an outline (32 bits).
s	indicates that the data type is a short integer (16 bits).
i	indicates that the data type is a long integer (32 bits).

Rectangles

The GL provides two types of rectangle subroutines—filled and unfilled. Filled rectangles are rectangular polygons, and unfilled rectangles are rectangular outlines. `rect()` draws a rectangular outline, while `rectf()` draws a filled (solid) rectangle. Only the *x* and *y* coordinates of the corners of the rectangle are given, and the *z* coordinate is forced to zero. The rectangle is aligned with the *x* and *y* axes.

Table 2-3 lists the six different forms of the rectangle subroutine.

Argument Type	Filled	Unfilled
16-bit integer	`rectfs()`	`rects()`
32-bit integer	`rectfi()`	`recti()`
32-bit float	`rectf()`	`rect()`

Table 2-3 Rectangle Subroutines

The arguments to the rectangle subroutines are the coordinates of the corners (*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 aligned with the axes, the coordinates of the other corners would be *(x1, y2)* and *(y1, x2)*.

Rectangles can undergo transformations as described in Chapter 7, "Coordinate Transformations," 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, then viewing it in perspective. On the screen, the rotated rectangle appears to be a trapezoid.

It is important to understand that although rectangles created with `rectf()` or its variants can be transformed by the statements described in Chapter 7, primitives such as rectangles, circles, arcs, and character strings are planar, 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 to proceed is to use the 3-D drawing functions `bgnpolygon()` and `endpolygon()`.

The following sample program, *chessboard.c* draws a chess board with black and white squares and red outlines on a green background.

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

#define ODD(n)   ((n) % 2)

main()
{
    int i, j;

    prefsize(400, 400);
    winopen("chessboard");
    color(GREEN);
    clear();
    for (i = 0; i < 8; i++) {
        for (j = 0; j < 8; j++) {
            if (ODD(i + j))
                color(WHITE);
            else
                color(BLACK);
            sboxfi(100 + i*25, 100 + j*25, 124 + i*25, 124 + j*25);
        }
    }
    color(RED);
    recti(97, 97, 302, 302);
    sleep(10);
    gexit();
    return 0;
}
```

Screen Boxes

Screen boxes are a subclass of rectangles. They are always 2-D and are always aligned with the screen coordinates. Draw screen boxes with `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 use 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 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.

Table 2-4 lists the screen box subroutines.

Argument Type	**Filled**	**Unfilled**
16-bit integer	`sboxfs()`	`sboxs()`
32-bit integer	`sboxfi()`	`sboxi()`
32-bit floating point	`sboxf()`	`sbox()`

Table 2-4 Screen Box Subroutines

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

Circles

Like rectangles, circles are 2-D figures that lie in the *x*-*y* plane, with *z* coordinates equal to zero. All six circle subroutines have the same parameters: *x*, *y*, and *radius*. Like rectangles, circles are either filled or unfilled, and the center coordinates and radius are specified in integers, short integers, or floats.

Table 2-5 lists the circle subroutines.

Argument Type	**Filled**	**Unfilled**
16-bit integer	`circfs()`	`circs()`
32-bit integer	`circfi()`	`circi()`
32-bit floating point	`circf()`	`circ()`

Table 2-5 Circle Subroutines

Circles are drawn with 80 equally spaced vertices, either as a closed line (unfilled circles) or as a polygon (filled circles). If your program draws many circles, you can write a circle primitive that uses fewer line segments to speed up the drawing. Circles drawn with 80 segments look reasonably good over a wide range of sizes, but on large circles, you can easily see the straight line segments. You can also draw circles with NURBS, as explained in Chapter 14.

This sample program, *bullseye.c*, draws an archery target using filled circles:

```
#include <gl/gl.h>
main()
{
    prefsize(400, 400);
    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(10);
    gexit();
    return 0;
}
```

Arcs

Arcs are also 2-D figures, and like circles and rectangles, they lie in the plane z=0. Arcs can be either unfilled (segments of circles) or filled (pie wedges). Arcs have a center *(x, y)*, a *radius*, a *starting angle*, and an *ending angle*. Angles are measured counterclockwise from the positive x axis; negative angles are clockwise. Angles are in tenths of degrees, so a 90-degree angle is written as 900.

An arc is drawn from the starting angle to the ending angle, so if *startang* is 0 and *endang* is 100, a 10 degree arc is drawn. Arcs are approximated by straight lines, and a full 360 degree arc consists of 80 segments. You can speed up arc drawing by making an arc primitive that uses fewer line segments.

Table 2-6 lists the `arc()` subroutines.

Argument Type	Filled	Unfilled
16-bit integer	`arcfs()`	`arcs()`
32–bit integer	`arcfi()`	`arci()`
32-bit float	`arcf()`	`arc()`

Table 2-6 Arc Subroutines

The following sample program draws a pie chart using filled arcs:

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

main()
{
   prefsize(400, 400);
   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(10);
   gexit();
   return 0;
}
```

2.2.2 Nonrecommended Old-Style Subroutines

This section describes *nonrecommended* subroutines from previous releases of the GL. Skip this section if you are developing new GL applications.

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, because the remaining subroutines are usually three-dimensional, they come in 2-D and 3-D 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.

Current Graphics Position

In the new architecture, graphical figures are sent together—a set of points, a polyline, and a polygon are sent bracketed by a `bgn`*`geometry`* and an `end`*`geometry`* subroutine. The drawing of the figure might not start until the `end`*`geometry`* 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 a polygon draw subroutines, and finally a polygon close subroutine.

Between the old-style subroutines drawing polylines or polygons, the system maintains 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. The current graphics position as used by the old-style polygon subroutines is discussed in the next sections.

getgpos

Because the system automatically maintains the current graphics position, few applications need to access it directly. Those that do use `getgpos()` to return 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.

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.

Old-Style Points

Table 2-7 shows the old-style point subroutines.

Argument Type	2-D	3-D
16-bit integer	`pnt2s()`	`pnts()`
32-bit integer	`pnt2i()`	`pnti()`
32-bit float	`pnt2()`	`pnt()`

Table 2-7 Old-Style Point Subroutines

The arguments are (*x*, *y*) for the 2-D subroutines, (*x*, *y*, *z*) for the 3-D subroutines. In addition to drawing a point, `pnt()` updates the current graphics position to its location.

This sample program, *pointsquare.c*, draws 100 points in a square area of the window:

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

main()
{
   int i, j;

   prefsize(400, 400);
   winopen("pointsquare");
   color(BLACK);
   clear();
   color(GREEN);
      for (i = 0; i < 100; i++) {
         for (j = 0; j < 100; j++)
            pnt2i(i*4 + 1, j*4 + 1);
   }
   sleep(10);
   gexit();
   return 0;
}
```

Old-Style Lines

Old-style lines are drawn using two subroutines: `move()` and `draw()`. The arguments and types of the `move()` and `draw()` subroutines are the same as for the `point()` subroutines. The `move()` subroutine sets the current graphics position to the specified vertex and `draw()` draws from the current graphics position to the specified point, then updates the current graphics position to that vertex.

Table 2-8 lists the `move()` and `draw()` subroutines.

Argument Type	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 2-8 Old-Style Move and Draw Subroutines

This sample program, *bluebox.c*, draws the outline of a blue box on the screen using the `move()` and `draw()` subroutines:

```
#include <gl/gl.h>
main()
{
   prefsize(400, 400);
   winopen("bluebox");
   color(BLACK);
   clear();
   color(BLUE);
   move2i(200, 200);
   draw2i(200, 300);
   draw2i(300, 300);
   draw2i(300, 200);
   draw2i(200, 200);
   sleep(10);
   gexit();
   return 0;
}
```

Old-Style Polygons

The old-style subroutines that draw filled polygons corresponding to the `move()` and `draw()` subroutines are `pmv()` and `pdr()`.

Table 2-9 lists the filled polygon subroutines.

Argument Type	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 2-9 Old-Style Filled Polygon Move and Draw 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.

Caution: Be sure not to spell the `pclos()` command *pclose*, because *pclose* is the IRIX command to close a pipe.

The following sample program, *bluebox3.c*, draws a filled blue polygon:

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

main()
{
   prefsize(400, 400);
   winopen("bluebox3");
   color(BLACK);
   clear();
   color(BLUE);
   pmv2i(200, 200);
   pdr2i(200, 300);
   pdr2i(300, 300);
   pdr2i(300, 200);
   pclos();
   sleep(10);
   gexit();
   return 0;
}
```

The GL has two sets of subroutines that take arrays of vertex coordinates and draw filled and unfilled polygons. Filled polygons are drawn by `polf()`, and polygon outlines are drawn by `poly()`.

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.

Table 2-10 lists the polygon and filled polygon subroutines.

Argument Type	**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 2-10 Old-Style Polygon and Filled Polygon Subroutines

This sample program, *hexagon.c*, draws a filled hexagon using `polf()`:

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

long parray[6][2] = {
   {100, 0},
   { 0, 200},
   {100, 400},
   {300, 400},
   {400, 200},
   {300, 0}
};

main()
{
   prefsize(400, 400);
   winopen("hexagon");
   color(BLACK);
   clear();
   color(GREEN);
   polf2i(6, parray);
   sleep(10);
   gexit();
   return 0;
}
```

Chapter 3

Characters and Fonts

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

- Section 3.1, "Drawing Characters," describes how to draw characters.
- Section 3.2, "Creating a Font," tells you how to create and enable your own font and how to query the system for font information.

The GL font interface supports the rapid display of raster characters in user-selectable fonts. You can design and use your own fonts, or make use of the default fonts supplied with the system. Typefaces exist or can be created with a variety of point sizes and with a fixed or variable pitch.

The IRIS Font Manager™, described in the *Graphics Library Windowing and Font Library Programming Guide*, offers a more flexible way to display text in a program than the method presented in this chapter. You can use the IRIS Font Manager to access externally created fonts, such as bitmap fonts, screen fonts, and X Window System fonts like *Portable Compiler Format* (PCF) fonts. See the *Graphics Library Windowing and Font Library Programming Guide* for information about loading and using externally created fonts and other facilities available through the IRIS Font Manager.

The font information in this chapter is provided mainly for performance reasons. By using the techniques presented in this chapter, you are using methods close to the hardware level of the system. Performance is as much as four times faster than is possible using the IRIS Font Manager. To get your application to display characters as rapidly as possible, use the techniques presented in this chapter.

There are two versions of the font definition and character drawing routines. The older version uses shorts as parameters and is thus limited to a character set that can only store 256 bitmap definitions and can handle only single-byte character data. Newer subroutines that use long integers, signified by the letter *l* in the subroutine name, can accommodate multibyte data and a more extensive index space.

Because the long integer subroutines offer a more flexible interface, their use is encouraged.

3.1 Drawing Characters

Use `cmov`() to position text and `charstr()`/`lcharstr()` to draw text. `cmov()` determines where the system draws text on the screen, `charstr()` draws a string of single-byte characters, and `lcharstr()` draws a string of multibyte characters.

The character string is drawn in the current color and in the current font. *Scaling*, *rotating*, or *translating* characters, operations that are described in Chapter 7, "Coordinate Transformations," has no effect on them. For example, when a geometry that has text labels associated with it shrinks as it moves away from the viewer, its labels remain the same size. Similarly, no matter what rotation is in effect, the character string maintains the same orientation, which is horizontal for any standard font.

3.1.1 Determining the Current Character Position

The *current character position* determines where the system draws text on the screen. `cmov()` moves the current character position to a specified point (*x*, *y*, *z*) in world coordinates. `cmov()` turns the world coordinates into window coordinates for the new character position. `cmov()` does not draw anything—it simply sets the character position where drawing is to occur when `charstr()` is issued. `cmov()` does not affect the current graphics position.

The current character position is *transformed* (see Chapter 7) in exactly the same way as a vertex to position a character string where it belongs.

Table 3-1 lists the `cmov()` subroutines.

Argument Type	2-D	3-D
Short integer	`cmov2s()`	`cmovs()`
Long integer	`cmov2i()`	`cmovi()`
Float	`cmov2()`	`cmov()`

Table 3-1 `cmov()` Subroutines

Note: Character position includes *z* values as well as *x* and *y* values. Because of this, you can associate string origins with 3-D locations and you can use z-buffering, described in Chapter 8, and depth-cueing, described in Chapter 13, with characters.

3.1.2 Drawing Character Strings

Use `charstr()` to draw a string of raster characters. Use `lcharstr()` to draw a character string using a long integer raster font—that is, characters that have been defined with `deflfont()`.

The text string is drawn in the current font using the current color. 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 ANSI C.

3.1.3 Clipping Character Strings

If the origin of a character string lies outside the *viewport* (see Chapter 7), no characters in the string are drawn.

If the origin of a character string is inside the viewport, the characters are individually *clipped* to the *screenmask*. A screenmask establishes a rectangular boundary on the screen. Clipping means that characters inside the boundary are displayed and characters outside of the boundary are not displayed.

Figure 3-1 shows how characters are clipped to the viewport and screenmask.

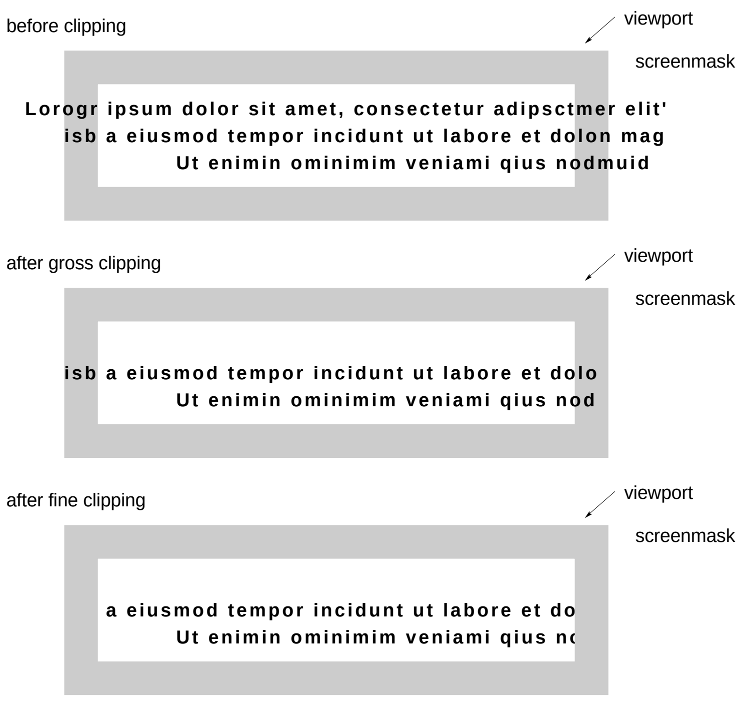

before clipping
viewport
screenmask
Lorogr ipsum dolor sit amet, consectetur adipsctmer elit'
isb a eiusmod tempor incidunt ut labore et dolon mag
Ut enimin ominimim veniami qius nodmuid
after gross clipping
viewport
screenmask
isb a eiusmod tempor incidunt ut labore et dolo
Ut enimin ominimim veniami qius nod
after fine clipping
viewport
screenmask
a eiusmod tempor incidunt ut labore et do
Ut enimin ominimim veniami qius nc

Figure 3-1 How Character Strings are Clipped

In *gross clipping*, character strings that appear outside the viewport are clipped out (not displayed).

There is a "gray area" between the screenmask and the viewport when the viewport is larger than the screenmask. In *fine clipping*, character strings that begin inside the viewport are clipped to the screenmask.

3.1.4 Getting Character Information

Use `getcpos()` to return the screen coordinates of the current character position into the locations pointed to by *ix* and *iy*:

```
void getcpos(short *ix, short *iy)
```

Use `strwidth()/lstrwidth()` to return the width of a text string in pixels:

```
long strwidth(String str)
```

```
long lstrwidth(long type, void *str)
```

The string can be any null-terminated ASCII string of characters. The value returned does not necessarily represent the width of a character times the number of characters in the string, because in some fonts, the character width varies from one character to another. The default font has a fixed width of nine pixels, so for that font, `strwidth()` does return nine times the string's length.

The *rasterchars.c* sample program draws two lines of text.

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

main()
{
   prefsize(400, 400);
   winopen("rasterchars");
   color(BLACK);
   clear();
   color(RED);
   cmov2i(50, 80);
   charstr("The first line is drawn ");
   charstr("in two parts. ");
   cmov2i(50, 80 - 14);
   charstr("This line is 14 pixels lower. ");
   sleep(10);
   gexit();
   return 0;
}
```

The first line of text in *rasterchars.c* is drawn in two parts. The first `cmov2i()` sets the current character position to 50 pixels to the right and 80 pixels up from the lower-left corner of the window. After the first string is drawn, the current character position is automatically advanced to follow the space character at the end of the line. When the character string "`in two parts.`" is drawn, it continues from the current character position. Next, the character position is set to start 14 pixels below the beginning of the top line, and the second line is drawn.

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 default font (font 0), which is defined when `winopen()` is called.

The next sample program, *rasterchars2.c*, shows that character strings are drawn in the same orientation no matter where they move, and that the current character position is transformed like any other geometry.

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

float p[3][2] = {
   {0.0, 0.0},
   {0.6, 0.0},
   {0.0, 0.6}
};

main()
{
   int i;

   prefsize(400, 400);
   winopen("rasterchars2");
   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(p[0]);
         v2f(p[1]);
         v2f(p[2]);
      endpolygon();
      color(GREEN);
      cmov2(p[0][0], p[0][1]);
      charstr("vert0");
      cmov2(p[1][0], p[1][1]);
      charstr("vert1");
      cmov2(p[2][0], p[2][1]);
      charstr("vert2");
      sleep(1);
   }
   gexit();
   return 0;
}
```

The output of *rasterchars2.c* shows a red triangle with its three vertices labeled *vert1*, *vert2*, and *vert3*. As the triangle rotates about *vert1*, the labels *vert2* and *vert3* move along with the triangle is if they were pinned to their respective vertices. The labels are drawn horizontally, no matter what position the triangle is in.

The `rotate()` subroutine rotates the scene about the *z* axis (coming directly out of the screen) by 5 degrees each time. The rotation is about the origin, so vertex *p1* remains fixed. See Chapter 7 for a description of `rotate()`.

3.2 Creating a Font

A font is a collection of rectangular arrays of *masks*. Masks determine whether or not a pixel is turned on. If a 1 appears in a mask, the corresponding pixel is turned on to the current color; if a 0 appears, the pixel is left as it is. For example, the following bitmask 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
```

Figure 3-2 Bitmask for the Character A

A font made up of single-byte characters 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 example, the ASCII value of A is 65 (decimal), so entry 65 in the font is associated with the bitmask for A shown in Figure 3-2. If this font were defined, the string “AAA” would draw three copies of the character whose bitmask appears in Figure 3-2.

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, because all bitmask data comes in 16-bit words. In the letter A example in Figure 3-2, the width is 12 and the height is also 12.

Normally, a character’s origin is at the lower-left corner of the bitmask, as is the case for the A. The character is drawn by placing its bitmask so that the bitmask’s origin is at the current character position.

Figure 3-3 shows a sample character definition for the letter g.

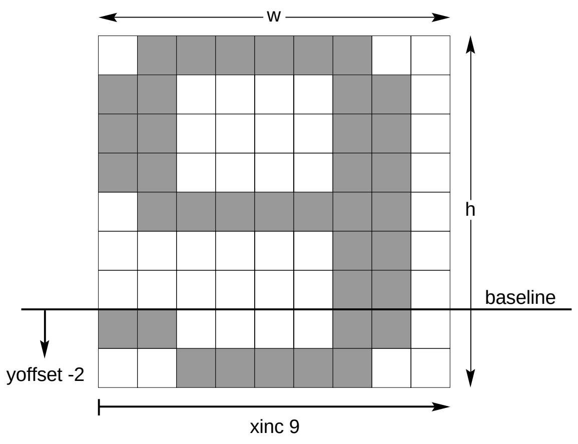


Figure 3-3 Character Bitmap Created with `defrasterfont()`

To define a single character like the one in Figure 3-3, you need the bitmask itself, *width*, *height*, *xoffset*, *yoffset*, and *xincrement*. The `defrasterfont()` and `deflfont()` subroutines allow you to define a collection of such characters.

For a character with a descender, such as g, j, or y, the two bottom lines of the bitmask should 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 3-3).

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 simplify matters, the character bitmasks are packed together in one array of 16-bit values, so the bitmask is determined by the offset into the bitmask array.

For example, if a single-byte font contains the letter A from Figure 3-2 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). The length and width together determine the number of shorts in a character's definition.

3.2.1 Defining the Font

There are two different subroutines for defining a font: `defrasterfont()` and `deflfont()`.

The `defrasterfont()` command is limited to 256 bitmap definitions within the raster array and uses `charstr()` to operate on the input string as a stream of single-byte character data.

Extended character sets require the use of *multibyte* character data, which is supported by `deflfont()`. The interface for using multibyte character sets is analogous to the single-byte interface, but is capable of containing more font information. This interface was designed primarily to support international fonts; however, you may find it useful for other purposes as well.

Using Single Byte Character Data

Use `defrasterfont()` to define a single-byte raster font. 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 of the default characters fit on the screen.

The ANSI C specification for `defrasterfont()` is:

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

where:

n	is an index into a font table. Font 0 is the default font; it cannot be redefined.
ht	is the maximum height of the font characters in pixels.
nc	is the number of elements in the *chars* array. The first 32 entries of *chars* are usually undefined because they correspond to the ASCII control characters.
chars	is an array of character descriptions of type *Fontchar,* which is defined in the header file *gl.h*. 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 *x* of the current character position after drawing the character.

nr	is the number of 16-bit integers in *raster.*
raster	is a one-dimensional array of *nr* bitmask bytes, ordered from left to right, then bottom to top. Mask bits are left-justified in the character's bounding box.

The following code fragment contains the defining parameters for the character in Figure 3-3 and shows the data in location 724 of the *chars* array:

```
defrasterfont(n, ht, nc, chars, nr, rasters);
chars ['g'] = {724, 8, 9, 0, -2, 9 }
short rasterarry [] =  {...
                       ...
                       0x7E00, 0xC300, 0x0300, 0x0300,
                       0x7F00, 0xC300, 0xC300, 0xC300,
                       0x7E00,
                       ...
                       }
```

Using Multiple Byte Character Data

Extensive character sets such as Asian ideograms require more raster data and a larger index space than is provided by `defrasterfont()`. Use `deflfont()` for multibyte character sets. The `deflfont()` subroutine is similar to `defrasterfont()`, except that:

- Larger amounts of raster data are supported because the number of raster elements is a long integer instead of a short integer.
- Character movement in the *y*-direction can be specified, allowing vertical displacement of characters.
- Very large bitmaps of characters can be supported, because height and width are short integers, instead of single-byte unsigned chars.
- Characters can be defined with an index beyond the single-byte, 256-character limit.
- Additional characters can be registered with an existing raster font after its initial definition. That is, if a character is currently defined at a specific offset within the raster array, it can be replaced with a new/different character. The same set of operations applied with `defrasterfont()`

results in the removal of the entire font set; thus, a single character can be modified with far less overhead using `deflfont()`.

The ANSI C specification for `deflfont()` is:

```
void deflfont(short n,long nc,Lfontchar chars, long nr, unsigned short raster[])
```

where:

n	is the value to use as the identifier for this raster font. The default font is a fixed-pitch ASCII font with a height of 15, width of 9, character values 0 through 127 defined, and is specified by a font identifier of 0. Font 0 cannot be redefined.
nc	is the number of elements in the *chars* array.
chars	is an array of character description structures of type Lfontchar. One structure is required for each character in the font.

The Lfontchar structure is defined in *<gl/gl.h>* as:

```
typedef struct {
      unsigned long value;
      unsigned long offset;
      short w, h;
      short xoff, yoff;
      short xmove, ymove;
} Lfontchar;
```

value	is the integer value corresponding to this character. When value is encountered by `charstr()` or `lcharstr()` this character is drawn.
offset	is the element number of the raster at which the bitmap data for this character begins. Element numbers start at zero.
w	is the number of columns in the bitmap that contain set bits (character width).
h	is the number of rows in the bitmap of the character (including ascender and descender).
xoff	is the offset in bitmap columns between the start of the character's bitmap and the start of the character.

yoff	is the number rows between the character's baseline and the bottom of the bitmap. For characters with descenders this value is a negative number. For characters that rest entirely on the baseline, this value is zero.
xmove	is the pixel spacing for the character. This signed value is added to the x-coordinate of the current raster position after the character is drawn.
ymove	is the pixel spacing for the character. This signed value is added to the y-coordinate of the current raster position after the character is drawn.
nr	is the number of 16-bit integers in raster.
raster	is a one-dimensional array containing all the bitmap data for the characters in the font. The bitmap data for each character is a set of consecutive, 16-bit integers, comprising the bit mask for the character from left to right, bottom to top. For characters of width greater than 16, the rows of a bitmap span more than one array element; however, each new row in the character bitmap must start with its own array element. The number of 16-bit integers per row in a character's bitmap is (w+15)/16. The total number of 16-bit integers in a character's raster definition is h∗((w+15)/16). Bit 15 of each element is left-most when displayed. Bits that are 1 are drawn; bits that are 0 are masked.

Examples of how to use this interface are included in *4Dgifts/examples/intl*.

3.2.2 Selecting the Font

Use `font()` to select the font that the system uses for drawing a text string. Its argument is the font number assigned to the font built by `defrasterfont()` or `deflfont()`. This font remains the current font until you call `font()` to select another font.

The next sample program, *font.c*, 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. When characters are not defined, no error occurs, but nothing is drawn 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
#define efont_nr(   (sizeof efont_bits)/sizeof(unsigned short))

#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[efont_nc];
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,
};

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);

   prefsize(400, 400);
   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();
    return 0;
}
```

3.2.3 Querying the System for Font Information

The following subroutines return information about the current font: its index, the height of its characters, and the maximum descender for its characters.

Use `getfont()` to query for the index of the current raster font:

```
long getfont(void)
```

Use `getheight()` to query for the maximum height, in pixels, 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):

```
long getheight(void)
```

Use `getdescender()` to query for the longest descender in the current font. It returns the number of pixels that the longest descender extends below the baseline:

```
long getdescender(void)
```

Chapter 4

Display and Color Modes

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

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

4.1 Color Display

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

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

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

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

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

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

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

4.1.1 Bitplanes

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

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

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

Table 4-1 Bitplane Configurations of Silicon Graphics Workstations

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

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

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

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

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

4.1.2 Dithering

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

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

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

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

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

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

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

4.2 RGB Mode

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

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

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

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

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

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

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

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

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

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

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

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

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

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

4.2.1 Setting and Getting the Current Color in RGB Mode

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

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

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

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

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

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

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

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

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

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

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

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

4.3 Gouraud Shading

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

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

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

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

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

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

The interpolation is linear in all three components.

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

(0,20,100)					
(15,28,90)	(0,16,80)				
(30,36,80)	(15,24,70)	(0,12,60)			
(45,44,70)	(30,32,60)	(15,20,50)	(0,8,40)		
(60,52,60)	(45,40,50)	(30,28,40)	(15,16,30)	(0,4,20)	
(75,60,50)	(60,48,40)	(45,36,30)	(30,24,20)	(15,12,10)	(0,0,0)

Figure 4-1 Gouraud Shaded Triangle

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

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

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

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

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

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

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

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

When you run this program, notice how the colors change smoothly along each edge, and across the interior of the triangle. You can modify the colors at the triangle's vertices to see how they affect the picture. Typically, polygons do not have wildly different colors at their vertices as the example does.

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

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

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

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

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

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

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

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

    if (getgdesc(GD_BITS_NORM_SNG_RED) == 0) {
        fprintf(stderr, "Single buffered RGB not available on this machine\n");
        return 1;
    }
    prefsize(400, 400);
    winopen("cylinder");
    RGBmode();
    gconfig();
    ortho2(-1.5, 1.5, -1.5, 1.5);
    c3f(blackvec);
    clear();

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

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

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

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


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

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

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

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

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

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

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

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

4.4 Color Map Mode

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

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

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

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

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

Table 4-3 Default Color Map

For example, the call

color(GREEN);

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

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

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

To change color map entries, use `mapcolor()`. `mapcolor()` takes four arguments: an index into the color map, and the red, green, and blue components to load into the map for that index. The index is between 0 and the system's limit, and the red, green, and blue components are integers between 0 and 255.

The calling sequence is:

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

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

4.4.1 Setting the Current Color in Color Map Mode

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

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

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

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

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

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

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

#define GRAY 64
#define PINK 65

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

4.4.2 Getting Color Information in Color Map Mode

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

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

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

4.4.3 Gouraud Shading in Color Map Mode

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

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

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

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

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

main()
{
 int i;

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

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

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

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

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

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

4.4.4 Blinking

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

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

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

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

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

main()
{
    int i;

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

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

To terminate blinking and restore the original color, call `blink()` with *rate* = 0 where *i* specifies a blinking color map entry. To terminate blinking of all colors, call `blink()` with *rate* = -1. When you set *rate* to -1, the other parameters are ignored.

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

4.5 Onemap and Multimap Modes

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

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

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

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

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

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

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

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

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

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

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

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

4.6 Gamma Correction

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Chapter 5

User Input

This chapter tells you how to program your application to respond to input from a user. User input can occur thorough a variety of devices, including the keyboard, mouse and peripheral devices.

- Section 5.1, “Event Handling,” describes event handling mechanisms and discusses input models.
- Section 5.2, “Input Devices,” describes the types of input devices available and lists the values they report.
- Section 5.3, “Video Control,” tells you how to query and control video parameters.

5.1 Event Handling

In the previous chapters, the emphasis has been on drawing graphics. You probably want to do more than just watch graphics on the screen; the next step is to add user input capability. A *user interface* provides a mechanism for detecting user input, acting on the input received, and determining when user input has ceased.

An *input event* is generated when you click a mouse button or press a key on the keyboard. Input events and the system responses associated with them are the dialogue used to communicate with the computer through the user interface. *Event handling* is the process that manages this communication: including rules that govern what constitutes an input event, how event priorities are established, and how events are communicated back and forth between the user and the computer.

User input events on the IRIS are handled by the *X server* (see the X Window System documentation). You can set up user input using X facilities, using the GL, or using a combination of X and GL known as *mixed-mode*, or *mixed-model*, programming.

Working with X gives you the advantage of using X-based *toolkits* and *widgets* to create your user interfaces. The X Window System provides facilities for event management through the Xt routines for timed events and X event routines for queued events. The X input model provides more information than the GL for each event and allows for multiple server, screen and window communication. If you want to use Motif™, Athena widgets, Xt, or multiple server I/O, you *must* use the X input model. See the X Window System documentation for more information about structuring X input environments.

For mixed-model programming, see the `glresources()` man page, and the `GLX` group of man pages, all of which begin with the upper-case letters GLX.

A GL program can detect input events in two ways: *queueing* or *polling*. Queueing saves input events and places them in an event *queue*, a section of memory where the events are stored in the order received; the first event in the queue is the first event to get processed. It is just like waiting in line to buy movie tickets—the first person in line gets the first tickets.

Polling continually *samples* (checks) the device(s) for input. Polling is not very efficient because the system spends a lot of time listening for input that could be used for doing other tasks. It is possible for the system to miss an input event that happens when it isn't listening.

Queueing is the recommended way to manage user input. Queueing allows you to process rapid up-and-down button transitions and capture all momentary device state changes.

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 *input focus*.

Input is enabled for the window that has the cursor in it; that window currently has the input focus. In a multiple window system input occurs in the window that has input focus; it does not occur in any other windows that are open concurrently.

5.1.1 Queueing

You decide which devices to queue, and establish rules about what constitutes a state change, or *event*, for those devices.

Any change in the state of a device for which queueing is enabled generates an entry in the event queue. Each queue entry consists of the device number and the current value of the device. If the input device is a button, the value is either 1 (pressed) or 0 (released). If the device is a *valuator*, such as the *x* position of the mouse, its value is an integer that indicates the position of the device.

To use queueing:

1. Enable queueing for the selected device with `qdevice()`.
2. Check the queue for an event, using `qread()`, `qtest()` or `blkqread()`.
3. Perform the appropriate action for that event.
4. Return to the event loop
5. Disable event queueing when the event loop is exited.

For maximum efficiency, you should not put complicated redraw operations inside the event loop and you should provide some mechanism for emptying the queue when it gets full.

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. Use `qreset()` to empty the queue when it gets full.

By default, only the pseudo-devices `INPUTCHANGE` and `REDRAW` are queued.

An `INPUTCHANGE` event appears when you direct the input focus to a new window or to the background. When an `INPUTCHANGE` event occurs, the identifier of the window that has input focus is placed on the queue; a 0 is placed on the queue when input focus is removed.

A `REDRAW` event appears in the queue when you move or reshape a window, when part of a window is uncovered or when the display mode changes.

GL subroutines for processing user input are listed next, with a brief description of what the command does, followed by its ANSI C specification.

qdevice

`qdevice()` enables queueing for 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(Device dev);
```

unqdevice

`unqdevice()` disables the queueing of events from the specified 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(Device dev);
```

isqueued

`isqueued()` finds out whether or not a specific device is queued. `isqueued()` returns a Boolean value. TRUE indicates that the device is enabled for queuing; FALSE indicates that the device is not queued.

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

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(short qtype, short val);
```

You can use any one of the next three subroutines—`qtest()`, `qread()`, `blkqread()`—to check the event queue for an entry.

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(short *data);
```

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*/2.

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(short *data, n);
```

qreset

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

```
void qreset(void);
```

qgetfd

`qgetfd()` allows a GL program to use the IRIX system call *select* 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()` does not cause the program to block.

```
long qgetfd(void);
```

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*. You `tie()` one valuator to a button by making *v2* equal to 0. 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 untie a button from valuators by making both *v1* and *v2* equal to 0.

```
void tie(Device b, Device v1, Device 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(Device vx, Device 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. `attachcursor()`, like `blink()`, is not reset to the default when the program exits.

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.

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

By default, the cursor is on when a window is created.Use `getcursor()` to find out whether the cursor is visible. See Chapter 11, "Frame Buffers and Drawing Modes," for more information on `getcursor()`.

noise

Some valuators are noisy; that is, they report small fluctuations, indicating movement when no event has occurred. `noise()` allows you to set a lower limit on what constitutes an event. The value of a noisy valuator *v* must change by at least *delta* before the motion is recognized. `noise()` determines how 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(Device v, short delta);
```

5.1.2 Polling

Polling immediately returns the value of a device that is a button or valuator, regardless of which window has focus. For example, the statement `getbutton(LEFTMOUSE)` returns 1 if the left button of the mouse is down and 0 if it is up. Programs that use polling should watch for changes to input focus and adjust their behavior accordingly.

getvaluator

`getvaluator()` polls the status of a valuator. The argument to `getvaluator()` is a valuator device number *(val)* that reflects the current state of the device.

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

getbutton

`getbutton()` polls the status of a button, whether the button is queued or not. The argument to `getbutton()` is the number of the device you want to poll (*num*). `getbutton()` returns TRUE if the button is down, FALSE if it is up.

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

getdev

`getdev()` polls up to 128 valuators and buttons concurrently. 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.) The *vals* array returns the state of each device in its corresponding array location.

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

The following sample program, *input.c*, uses queueing to control a simple drawing program, that lets you sketch in the window with the left mouse button.

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

#define X    0
#define Y    1

main()
{
   short val, mval[2], lastval[2];
   long org[2], size[2];
   Device dev, mdev[2];
   Boolean run;
   int leftmouse_down = 0;
   lastval[X] = -1;
   prefsize(400, 400);
   winopen("input");
   color(BLACK);
   clear();
   getorigin(&org[X], &org[Y]);
   getsize(&size[X], &size[Y]);
   mdev[X] = MOUSEX;
   mdev[Y] = MOUSEY;
   getdev(2, mdev, lastval);     /* initialize lastval[] */
   lastval[X] -= org[X];
   lastval[Y] -= org[Y];
```

```
    qdevice(LEFTMOUSE);
    qdevice(ESCKEY);
    qdevice(MOUSEX);
    qdevice(MOUSEY);
    color(WHITE);          /* prepare to draw white lines */

  while (1) {
    switch (dev = qread(&val)) {
       case LEFTMOUSE:
          leftmouse_down = val;
          break;
       case MOUSEX:
          val[X] = val - org[X];
          break;
       case MOUSEY:
          mval[Y] = val - org[Y];
          if (leftmouse_down) {
             bgnline();
                v2s(lastval);
                v2s(mval);
             endline();
          }
          lastval[X] = mval[X];
          lastval[Y] = mval[Y];
          break;
       case ESCKEY:
          exit(0);
    }
  }
}
```

5.2 Input Devices

Input devices accept user input and translate that input into data that a program can use. The GL supports three classes of input devices:

Buttons	Return a Boolean value: FALSE when they are not pressed (open) and TRUE when they are pressed (closed).
Valuators	Return an integer value that represents their current status. For example, a mouse is a pair of valuators: one reports horizontal position and the other reports vertical position.
Pseudo-devices	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 1 to 32767, inclusive. Table 5-1 shows how the device domain is organized.

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

Table 5-1 Class Ranges in the Device Domain

Facilities exist to let you define your own input devices. Additional information is available in the GL man pages.

5.2.1 Buttons

User input buttons include the mouse buttons, keyboard keys, buttons on a dial and button box, digitizer tablet buttons, and a menu button. Table 5-2 lists the names of the buttons and their descriptions.

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 5-2 Input Buttons

5.2.2 Valuators

Valuators are single-value input devices. Valuators report a 16-bit integer value, such as the horizontal and vertical position of the mouse, or the current setting of a dial. Table 5-3 shows the valuator names and descriptions.

Devices	Description
MOUSEX	x valuator on mouse
MOUSEY	y valuator on mouse
DIAL0...DIAL7	Position of 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 5-3 Input Valuators

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

Timer Devices

The GL timer devices are used to get input events at regular intervals. The timers use an internal clock that approximates the screen refresh interval. The clock rate is approximately 67 Hz. The event “time” returned by the timers is the frame count recorded during the elapsed event. You should not use GL timers to measure chronological time, nor should you use them to synchronize your graphics programs. To record events less frequently, use `noise()`.

For example, if you call `noise (TIMER0, 30),` only every 30th event is recorded, generating one event approximately every half second.

Cursor Devices

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

Ghost Devices

Ghost devices, `GHOSTX` and `GHOSTY`, do not correspond to 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.

5.2.3 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.

Note: There is a hardware mechanism in the keyboard that reads multiple key-down events if the key is held down and begins to auto-repeat.

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),` 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 and the value 1 when the key is pressed; 0 when it is released.

5.2.4 Window Manager Tokens

The tokens listed in Table 5-4 can be queued as pseudodevices to monitor GL window manager events, which are generated when windows are activated or moved. In all cases, the system returns the window identification number (id) of the window experiencing the event.

Token	Description
`DEPTHCHANGE`	Indicates an open window has been pushed or popped. This token is not supported.
`DRAWOVERLAY`	Indicates damage to the overlay planes. Queue this token if you use overlay planes.
`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 id of the window that has just gained input focus.
`REDRAW`	The window manager inserts a `REDRAW` token each time the window needs to be redrawn. The `REDRAW` token is queued automatically.
`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.
`WINSHUT`	When queued, the window manager sends this token when Close 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 grayed out and has no effect if selected.
`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 multiwindow application.

Table 5-4 Window Manager Event Tokens

5.2.5 Spaceball™ Devices

Table 5-5 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 5-5 Spaceball Input Buttons

For more information about the optional Spaceball input device, see the documentation that is packaged with the Spaceball option.

5.2.6 Controlling Devices

The GL provides subroutines that initialize device values and control the characteristics and behavior of the system's peripheral input/output devices. For example, some of these routines turn the keyboard click on, `clkon()`, and off, `clkoff()`, or set the keyboard bell. You set these controls to your preference or needs.

setvaluator

`setvaluator()` assigns an initial value (*init*) to a valuator. The arguments *min* and *max* are the minimum and maximum values the device can assume.

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

clkon

If `clkon()` is called, the keyboard makes an audible click whenever a key is pressed.

```
void clkon(void);
```

clkoff

`clkoff()` turns off the keyboard click.

```
void clkoff(void);
```

lampon

`lampon()` and `lampoff()` control the four lamps on old-style keyboards labeled L1, L2, L3, and L4 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(Byte lamps);
```

lampoff

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

```
void lampoff(Byte lamps);
```

ringbell

`ringbell()` rings the keyboard bell.

```
void ringbell(void);
```

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(Byte 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(char str[8]);
```

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(unsigned long mask);
```

5.3 Video Control

You can query the status and control the behavior of certain video parameters. You can also determine information about the video monitor used on your system and specify its operating parameters with the following subroutines.

Note: RealityEngine systems use a graphical user interface to set the video format. See the RealityEngine Owner's Guide for information about this interface and the video formats available.

blankscreen

`blankscreen()` turns the screen display on and off. b=TRUE stops display; b=FALSE restarts display.

```
void blankscreen(Boolean bool);
```

blanktime

`blanktime()` sets the screen blanking time-out. 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.

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

nframes specifies the screen blanking time-out 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.

setmonitor

`setmonitor()` sets the monitor to one of the video formats listed in Table 5-6.

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

Not all formats are supported by all systems. If a format is not supported, it is ignored. Some formats may require the use of optional products.

Note: IRIS Indigo systems do not support `setmonitor()`. Elan supports NTSC, PAL, HZ60, HZ72, STR_RECT, and RS-343 monitors.

getmonitor

`getmonitor()` queries the type of the current display monitor, as listed in Table 5-6.

Table 5-6 lists monitor type tokens and the video formats they represent.

Type	Video Format
STR_RECT	120Hz stereo format, if supported by hardware
HZ90_STEREO	90Hz stereo format, if supported by hardware
HZ76	76Hz, noninterlaced
HZ72	72Hz, noninterlaced
HZ60	60Hz noninterlaced
HZ30	30Hz interlaced
HZ30_SG	30Hz noninterlaced with sync on green
A343	RS-343 component RGB (1280×960), if supported by hardware
HDTV	HDTV format, if supported by hardware
PAL	PAL- 625 line component RGB (768×575) or SECAM
NTSC	NTSC - RS1070A 525 line component RGB (768×575)
VGA	VGA component RGB (640×497)
PR60	Pixel replication of one-quarter resolution 60Hz format

Table 5-6 Monitor Types

getothermonitor

`getothermonitor()` returns the other monitor types supported by the hardware. Most systems are not limited to one optional video mode. The display hardware can support all the video modes. `getothermonitor()` normally returns `MON_ALL` showing that all monitor types are supported. `getothermonitor()` returns `MON_GEN_ALL` if the optional genlock board is installed in the system.

setvideo

`setvideo()` sets the specified video hardware register, *reg*, to the indicated *value*. `setvideo()` and `getvideo()` support several video boards. The DE_R1 is physically present on IRIS-4D/B/G/GT/GTX systems, and is emulated on other systems.

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, which indicates that *reg* is not a valid register, or that you queried a video register on a system without that particular board installed

Table 5-7 lists the video register values for `setvideo()` and `getvideo()`.

Video Option Board	Register
Display Engine Board	DE_R1
CG2/CG3 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

Table 5-7 Video Register Values

Video Option Board	Register
	VP_MAPBLUE
	VP_MAPGREEN
	VP_MAPRED
	VP_MAPSRC
	VP_MAPSTROBE
	VP_PIXCNT
	VP_SAT
	VP_STATUS0
	VP_STATUS1
	VP_VBLANK
	VP_WIDTH

Table 5-7 **(continued)** Video Register Values

videocmd

`videocmd()` initializes the Live Video Digitizer option. If you don't have a Live Video Digitizer, you might want to skip this section.You can initialize the Live Video Digitizer in either RGB or composite video mode, for both NTSC and PAL video sources. The *cmd* parameter initiates the specified command. Table 5-8 lists the values for *cmd,* which are defined in the file *gl/vp1.h*.

Token	Description
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

Table 5-8 Live Video Digitizer Commands

Token	Description
VP_INITPAL_RGB	Initialize the Live Video Digitizer for an RGB PAL video source

Table 5-8 Live Video Digitizer Commands

Chapter 6

Animation

This chapter describes the subroutines that you use to create continuous motion in graphics scenes. This chapter introduces a technique called *double buffering*, which allows you to create graphics that are animated in the kind of smooth motion that looks like a movie.

- Section 6.1, "Understanding How Animation Works," presents an overview on how animation is done on the IRIS.
- Section 6.2, "Creating an Animation," tells you how to set up your system to do animation, and maximize animation performance.

6.1 Understanding How Animation Works

You can address frame buffer memory in either of two modes:

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

The currently visible buffer is called the *front buffer* and the invisible, drawing buffer is the *back buffer*. 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 to 76 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.

6.1.1 Single Buffer Mode

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.

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.

6.1.2 Double Buffer Mode

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. Scene frames are interleaved in this manner so that the changes from one frame to the next cannot be detected by your eye. If the frames are not swapped quickly enough, your eye can detect this motion and perceive it as flicker.

In double buffer mode, the bitplanes are partitioned into two groups, the front bitplanes and the back bitplanes. Double buffering works in either RGB mode or color map mode. Use `doublebuffer()` to set the display mode to double buffer mode. The mode change does not take effect until you call `gconfig()`. In double buffer mode, only the front bitplanes are displayed, and drawing routines normally update only the back bitplanes; `frontbuffer()` and `backbuffer()` can override this default. `gconfig()` sets `frontbuffer()` to `FALSE` and `backbuffer()` to `TRUE` in double buffer mode.

6.2 Creating an Animation

Before drawing anything, you must set the frame buffers into the correct configuration, such as color map mode or RGB mode.

6.2.1 Setting Modes for Animation

Configuring the frame buffer is a two-step process:

1. Indicate how to configure the frame buffer.
2. Call `gconfig()` to set the system into that particular configuration.

After a `gconfig()` call, the current writemask and color are no longer defined.

6.2.2 Swapping Buffers

After the entire frame is rendered into the back buffer, `swapbuffers()` is called to make it the visible buffer. The `swapbuffers()` call is ignored in single buffer mode. The `swapbuffers()` subroutine waits for the next screen refresh before exchanging the front and back buffers. If it did not wait, a frame would be drawn partly in one buffer, and partly in the other, causing a serious visual disturbance.

Because screen refresh occurs approximately 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 to 76 times per second. Other monitor options can have other retrace periods. See `setmonitor()` and `getmonitor()` in Chapter 5.

Because `swapbuffers()` blocks the user program until the next screen refresh (1/60 second), 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. While the geometry is moving about, the time it takes to render each frame varies.

6.2.3 Swapping Multiple Buffers

On IRIS-4D/VGX Series, SkyWriter, and RealityEngine 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. Just like `swapbuffers()`, `mswapbuffers()` blocks until the next vertical retrace period. It is ignored by frame buffers that are not in double buffer mode.

The constants you use to indicate the buffers to be swapped 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 bitwise-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 this line in your program:

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

This sample program, *bounce.c*, demonstrates animation.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <gl/gl.h>
#include <gl/device.h>
#define RGB_BLACK 0x000000
#define RGB_WHITE 0xffffff
#define X           0
#define Y           1
#define XY          2
#define SIZE        0.05
#define BOUNDS      (1.0 + SIZE)
#define EDGE        (1.1 * BOUNDS)

float boundary[4][XY] = {
    {-BOUNDS, -BOUNDS},
    {-BOUNDS, BOUNDS},
    { BOUNDS, BOUNDS},
    { BOUNDS, -BOUNDS}
};


struct ball_s {
    float pos[XY];
    float delta[XY];
    unsigned long col;
};
```

```
void main(int argc, char *argv[])
{
    int i, j;
    int nballs;
    struct ball_s *balls;
    short val;
    if (getgdesc(GD_BITS_NORM_DBL_RED) == 0) {
        fprintf(stderr, "Double buffered RGB not available on this machine\n”);
        return 1;
    }
    if (argc != 2) {
        fprintf(stderr, "Usage: bounce <ball count>\n");
        return 1;
    }
    nballs = atoi(argv[1]);
    if (!(balls = (struct ball_s *)malloc(nballs * sizeof(struct ball_s)))) {
        fprintf(stderr, “bounce: malloc failed\n”);
        return 1;
    }
    for (i = 0; i < nballs; i++) {
        for (j = 0; j < XY; j++) {
            balls[i].pos[j] = 2.0 * (drand48() - 0.5);
            balls[i].delta[j] = 2.0 * (drand48() - 0.5) / 50.0;
        }
        balls[i].col = drand48() * 0xffffff;
    }
    prefsize(400, 400);
    winopen("bounce");
    doublebuffer();
    RGBmode();
    gconfig();
    shademodel(FLAT);
    qdevice(ESCKEY);
    ortho2(-EDGE, EDGE, -EDGE, EDGE);
    while (!(qtest() && qread(&val) == ESCKEY && val == 0)) {
        for (i = 0; i < nballs; i++) {
             for (j = 0; j < XY; j++) {
                    balls[i].pos[j] += balls[i].delta[j];
                    if ((balls[i].pos[j] >= 1.0) || (balls[i].pos[j] <= -1.0))
                        balls[i].delta[j] = -balls[i].delta[j];
                }
            }
            cpack(RGB_BLACK);
            clear();
            cpack(RGB_WHITE);
            bgnclosedline();
            for (i = 0; i < 4; i++)
                v2f(boundary[i]);
            endclosedline();
            for (i = 0; i < nballs; i++) {
                cpack(balls[i].col);
                sboxf(balls[i].pos[X]-SIZE, balls[i].pos[Y]-SIZE,
                        balls[i].pos[X]+SIZE, balls[i].pos[Y]+SIZE);
        }
        swapbuffers();
    }
    gexit();
    return 0;
}
```

6.2.4 Maximizing Animation Performance

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.

When the value of the argument *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. `frontbuffer(TRUE)` in double buffer mode enables simultaneous updating of (or writing into) both the front and the rear buffers. `gconfig()` sets `frontbuffer()` to `FALSE`.

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, *b*, 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.

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.

Each of the possible values also has an associated symbolic value that you can use in the call to `getbuffer()`. See the `getbuffer()` man page for a list of these symbols.

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 `swapbuffers()`. `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.

Note: `swapinterval()` is valid only in double buffer mode.

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.

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, but it is not precise since the retrace rate varies among systems. You might think of calling `gsync()` a number of times to get short time delays (because each call waits until the next 60th of a second), but this is not the recommended procedure for setting up a delay. Use the system call `sginap()` instead.

`gsync()` is also used for rubber-banding in single buffer mode.

`gsync()` is included primarily for compatibility with systems that might not have enough bitplanes to use double buffer mode. `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.

Chapter 7

Coordinate Transformations

This chapter describes the subroutines that you use to set up a graphics scene, for example, how much of the screen to use, where to locate the viewing point, where shapes in the scene are located in space, how much of the scene is visible, and what type of view you are looking at.

- Section 7.1 describes the coordinate systems that the GL uses and explains how they are derived.
- Section 7.2, “Projection Transformations,” describes how coordinates are mapped to the screen and tells you how to set up your scene projection.
- Section 7.3, “Viewing Transformations,” tells you how to set up the view of your scene.
- Section 7.4, “Modeling Transformations,” tells you how to locate a geometry in space and how to change its size and orientation.
- Section 7.5, “Controlling the Order of Transformations,” explains how to put the items in your scene where you want them to go and how to save a scene setup for later use.
- Section 7.6, “Hierarchical Drawing with the Matrix Stack,” tells you how to draw items that are grouped into hierarchies.
- Section 7.7, “Viewports, Screenmasks, and Scrboxes,” tell you how to define the visible limits of your scene.
- Section 7.8, “User-Defined Transformations,” tells you how to define your own way of looking at the scene and manipulating items in it.
- Section 7.9, “Additional Clipping Planes,” tells you how to define the visible limits of your scene using boundaries that you create.

Changing how your graphics scene is viewed and changing where items are placed in it are called *transformations*. Three-dimensional transformations are difficult to describe, and even harder to visualize. You may need to read the chapter more than once, study the illustrations, experiment with the sample code, and write your own programs in order to gain an understanding of these topics.

7.1 Coordinate Systems

You use many different coordinate systems in the process of drawing a graphics scene. Figure 7-1 illustrates the coordinate systems used at different stages of the drawing process.

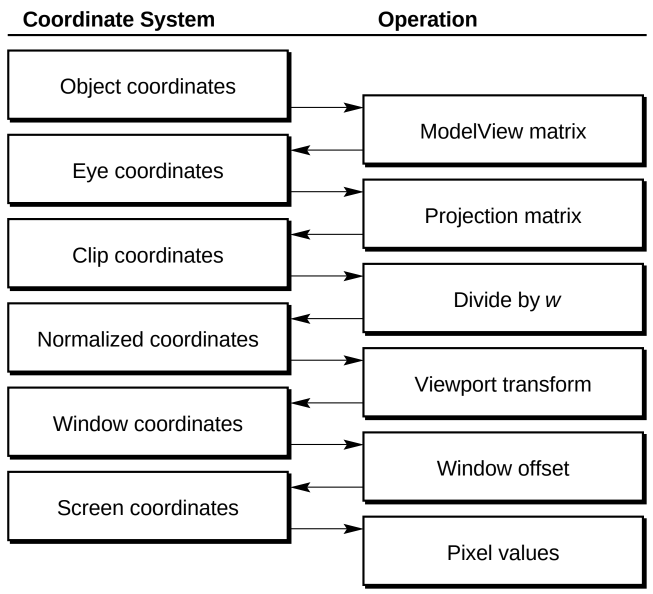


Figure 7-1 Coordinate Systems

The coordinate systems on the IRIS begin with a 3-D system defined in *right-handed cartesian floating point* coordinates, called the *object coordinate system*. Geometry vertices that you specify in *(x, y, z)* triplets are in this coordinate system. There are no limits to the size of coordinates in this system (other than the largest legal floating point value).

The *eye coordinate system* is the result of *transforming* (through matrix multiplication) the geometry coordinates by the contents of the modeling and viewing (*ModelView*) matrix. The eye coordinate system is the system in which lighting calculations are performed internally.

When the IRIS transforms points expressed in eye coordinates by the *Projection* matrix, the output is expressed in *clip coordinates*. Values returned by a call to `getgpos()` are expressed in this coordinate system.

The next system is called the *normalized coordinate system*. 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 3D unit cube are scaled directly into the next coordinate system, usually called the *window coordinate system*. The pixel at the lower-left corner of a window has window coordinates (0,0).

Window coordinates, 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 2-D, but in fact all three dimensions of the normalized coordinates are scaled, and there is a screen z coordinate that can be used for hidden surface removal, described in Chapter 8, or depth cueing, described in Chapter 11.

Note: 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 into a single matrix (ModelView), there are no world coordinates in the GL.

To get from geometry vertices to a scene displayed on your screen, you have to specify a *projection transformation*, to define how images in your scene are projected to the screen and a *viewing transformation*, to define what type of view you have and where you are viewing the scene from. This is sort of like setting up a camera to look through at the scene. You can move the camera around and change lenses to get different views with these transformations.

7.2 Projection Transformations

You can project an image onto the screen in one of three ways:

- Perspective - Items far away are smaller than items close to you, and parallel lines appear to recede into the distance toward a vanishing point.

  This is how you see the real world through your eye's ability to perceive depth. Consequently, scenes with a perspective projection will look and feel more natural to you, unless you have exaggerated some parameter that causes distortion.

- Window - Items are seen in perspective, but it is possible to create an asymmetric view of the scene.
- Orthographic - Items are projected through a rectangular viewing volume, but are not seen in perspective.

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 in Section 7.7, "Viewports, Screenmasks, and Scrboxes"). The viewpoint is where your eye is with respect to the viewing volume. This is called the *eye position*, or simply the *eye*.

Projection transformations are either perspective or orthographic in nature. The difference between the three projection transformations is the definitions of their *viewing volumes*. Perspective projections create a volume that has the shape of a pyramid with the top cut off. Orthographic projections create a viewing volume that has parallel sides.

7.2.1 Perspective Projection

Viewing items in perspective on the computer screen is like looking through a rectangular piece of perfectly transparent glass. Imagine drawing a line from your eye through the glass until it hits the item, coloring a dot on the glass where the line passes through the same color on the item. If this were done for all possible lines through the glass, if the coloring were perfect, and if the eye not allowed to move, 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 block your view of the portions of the items outside the glass. 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 or too near to be seen clearly with the eye, just like your eye cannot focus on objects that are very far away or very close to you. These 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 geometries 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, as discussed in Chapters 8 and 11.

For a perspective view, the visible region of the *world* (your graphics scene) looks like a pyramid with the top sliced off. The technical name for this is a *frustum*, or *rectangular viewing frustum*.

In a perspective projection, the Projection matrix maps a frustum of eye coordinates so that it exactly fills the unit cube (after *x*, *y*, and *z* are each divided by w). This frustum 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 perpendicular to the line of sight.

Use `perspective()` to define a perspective projection:

```
void perspective(Angle fovy, float aspect, Coord znear, Coord zfar)
```

`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 (*fovy*) is an angle made by the top and bottom clipping planes that is measured in tenths of degrees, so a 90 degree angle is specified as 900.

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 the aspect ratio 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 `keepaspect()` subroutine tells the window manager to maintain the window's *x* and *y* dimensions in a 1 to 1 ratio, that is, 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 want to try this to see how it looks. 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. Use `getsize(x,y)` to return the height and width in pixels of a GL window.

Figure 7-2 shows a frustum that demonstrates eyepoint, FOV angles, clipping planes, and aspect ratio.

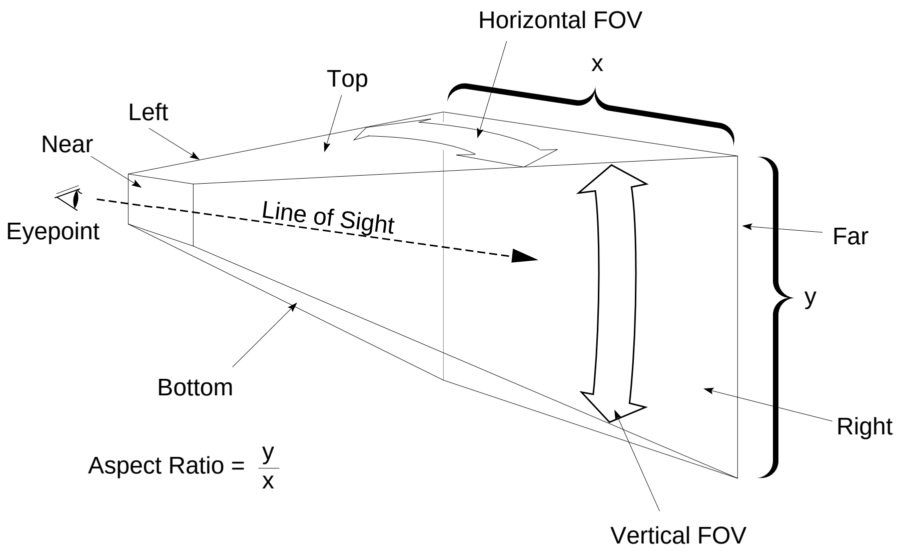


Figure 7-2 Frustum

This sample program, *perspective.c,* draws a single rectangle whose shade varies from bright red at z=0.0 to bright green at z=-4.0.

```
#include <stdio.h>
#include <gl/gl.h>
#define RGB_BLACK 0x000000
#define RGB_RED 0x0000ff
#define RGB_GREEN 0x00ff00

float v[4][3] = {
    {-3.0, 3.0, 0.0},
    {-3.0, -3.0, 0.0},
    { 2.0, -3.0, -4.0},
    { 2.0, 3.0, -4.0}
};

main()
{
    long xsize, ysize;
    float aspect;

    if (getgdesc(GD_BITS_NORM_SNG_RED) == 0) {
        fprintf(stderr, “Single buffered RGB not available\n”);
    return 1;
    }
    prefsize(400, 400);
    winopen("perspective");
    mmode(MVIEWING);
    getsize(&xsize, &ysize);
    aspect = (float)xsize / (float)ysize;
    perspective(900, aspect, 2.0, 5.0);
    RGBmode();
    gconfig();
    cpack(RGB_BLACK);
    clear();
    bgnpolygon();
        cpack(RGB_RED);
        v3f(v[0]);
        v3f(v[1]);
        cpack(RGB_GREEN);
        v3f(v[2]);
        v3f(v[3]);
    endpolygon();
    sleep(10);
    gexit();
    return 0;
}
```

Figure 7-3 shows a top view of the scene drawn in the sample program *perspective.c*. This view is not the view you see on your screen when you run the program; this view lets you see outside of the viewing volume.

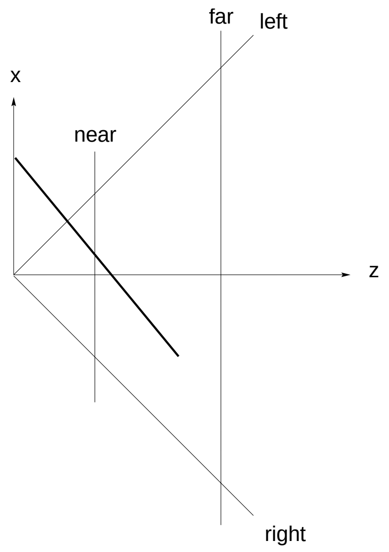


Figure 7-3 Clipping Planes

The heavy line shows the rectangle that the program draws. The eye is at (0.0, 0.0, 0.0), so 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.

The visible part of the polygon nearest the eye is not bright red, but already looks yellowish because the rectangle is clipped by the near clipping plane. If you bring in the near clipping plane toward you, the near end of the polygon looks more and more red.

7.2.2 Window Projection

A `window()` projection transformation shows the world in perspective view. 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(Coord left,Coord right,Coord bottom,Coord top,Coord near,Coord far)
```

Figure 7-4 illustrates the window projection transformation.

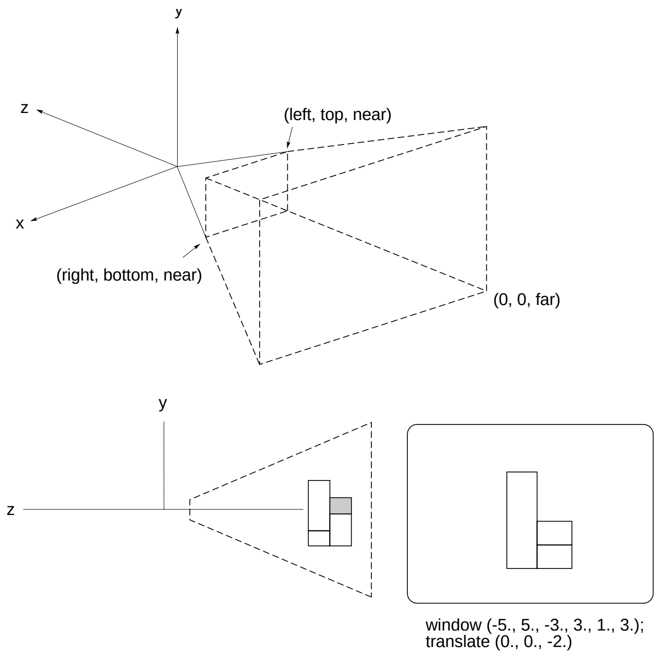


Figure 7-4 The `window()` Projection Transformation

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, described in Chapter 15.

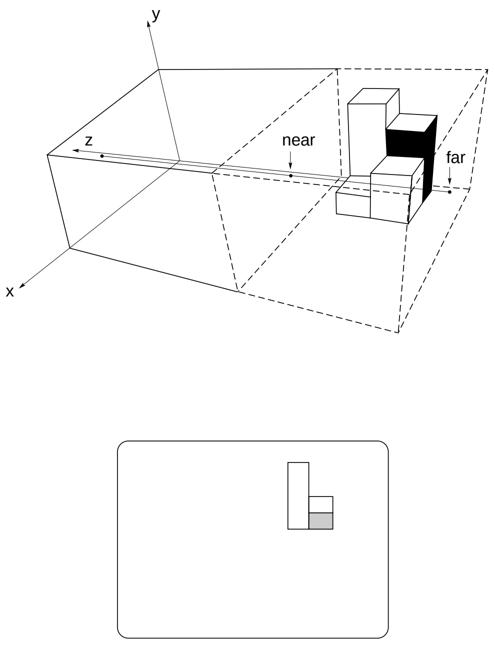
`window()` specifies the position and size of the rectangular viewing frustum closest to the eye, the location of the near and far clipping planes. `window()` projects a perspective view of the image onto the screen.

7.2.3 Orthographic Projections

The remaining two projection subroutines are the orthographic transformations, `ortho()` and `ortho2()`. 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, then the geometry inside is projected parallel to the *z* axis onto a face parallel to the *x*-*y* plane.

Figure 7-5 shows an example of a 3D orthographic projection.



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

Figure 7-5 The `ortho()` Projection Transformation

`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 -1 and 1.

`ortho2()` is usually used for 2-D drawing, where all the z coordinates are zero. It is really a 3D transformation; if you use `ortho2()` and try to draw objects with *z* coordinates outside the range $-1.0 \leq z \leq 1.0$, they are clipped out.

ortho

`ortho()` defines a box-shaped enclosure in the eye coordinate system:

```
void ortho(Coord left,Coord right,Coord bottom,Coord top,Coord znear,Coord zfar)
```

The arguments *left*, *right*, *bottom*, and *top* define the *x* and *y* clipping planes. *znear* and *zfar* are distances along the line of sight and are negative. In other words, the *z* clipping planes are located at *z* = –*znear* and *z* = –*zfar*.

Note: Because `ortho()` allows separate specification of the left, right, bottom, and top clipping planes rather than just the width and height of the viewing volume, it can move the effective viewpoint off the positive *z* axis. Thus, although `ortho()` is a projection transformation, it exhibits some aspects of a viewing transformation. Be careful when using `ortho()` with asymmetric volume boundaries so that you do not duplicate this viewpoint offset in your viewing transformation.

ortho2

`ortho2()` defines a 2-D clipping rectangle:

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

The arguments *left*, *right*, *bottom*, and *top* define the sides of the rectangle. Choose the values for `ortho2()` carefully, 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 pixel centers rather than on pixel boundaries. You need to add a .5 to each value in the `ortho2()` call, so that pixels are centered on whole numbers. If you do not do this, you might find that accumulated round-off errors in the arithmetic of pixel operations cause your pixel specifications to be off by a pixel or more.

The `ortho2()` statement in the following code fragment defines the correct clipping rectangle for the window created with the corresponding `prefposition()` statement:

```
prefposition(101, 500, 101, 500);
ortho2(100.5, 500.5, 100.5, 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.

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 and 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. The eye position is specified by a radius (distance from the origin) and by angles measuring the azimuth and elevation. The specification is similar to a polar coordinates. There is still one degree of freedom, because these values tell only where the eye is relative to the object.

`lookat()` allows you to specify both the viewpoint and the reference point toward which the eye is looking, and a *twist* angle to specify which way is up.

Both viewing subroutines work with a projection subroutine. If you want to view a point (for example: 1, 2, 3) from another point (4, 5, 6) in a perspective view, use both `perspective()` and `lookat()`.

When the orthographic 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, with rotations and translations, the position of the eye to the origin and the viewing direction so that it lies along the negative *z* axis.

7.3.1 Viewpoint in Polar Coordinates

`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 origin in world coordinates. *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 origin (0,0,0).

The *twist* rotates the viewpoint around the line of sight using the *right-hand rule* (as you look down the positive rotation axis to the origin, positive rotation is counterclockwise). All angles are specified in tenths of degrees and are integers.

Figure 7-6 shows examples of `polarview()`.You don't see anything in the top picture because the eye is positioned on the origin, looking at the origin.

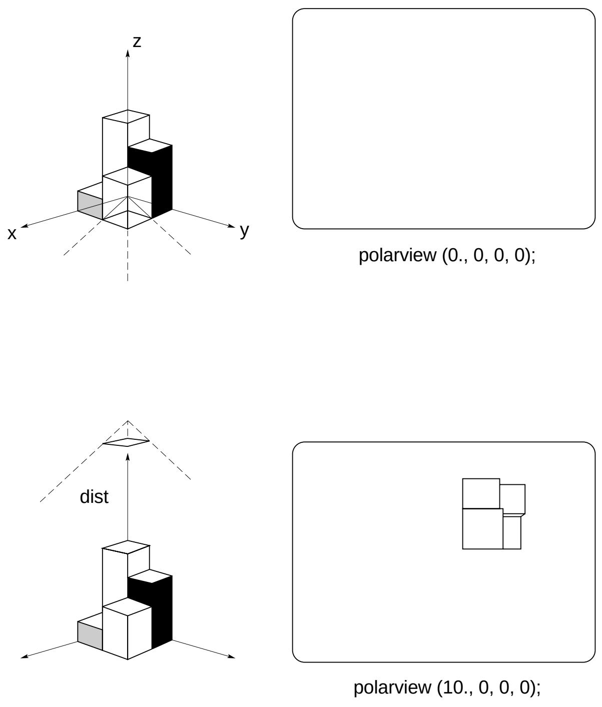


Figure 7-6 The `polarview()` Viewing Transformation

7.3.2 Viewpoint along a Line of Sight

`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. The *twist* measures right-hand rotation about the *z* axis in the eye coordinate system.

Figure 7-7 shows examples of `lookat()`.

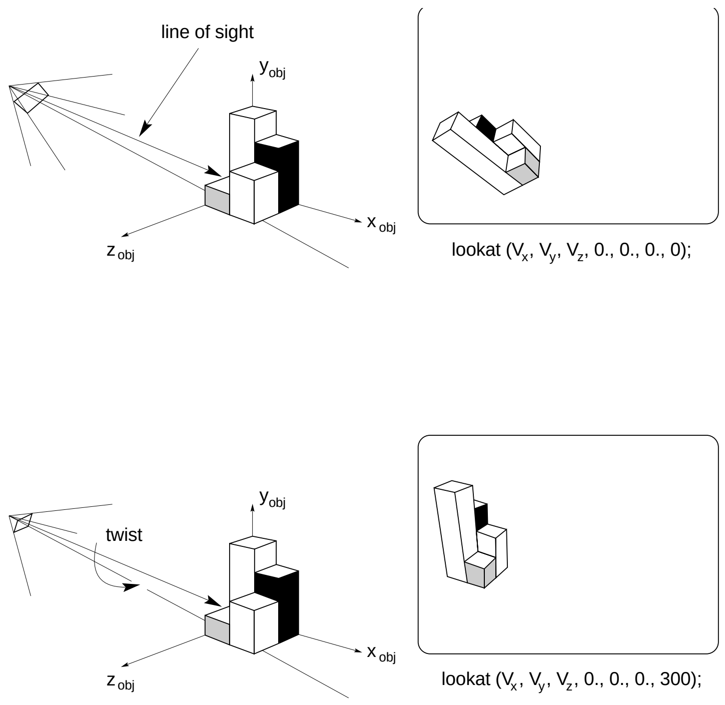

line of sight
y_{obj}
x_{obj}
z_{obj}
lookat (V_x, V_y, V_z, 0., 0., 0., 0);
y_{obj}
twist
x_{obj}
z_{obj}
lookat (V_x, V_y, V_z, 0., 0., 0., 300);

Figure 7-7 The `lookat()` Viewing Transformation

The *lookat.c* sample program draws a wireframe cube centered at the origin. Its 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 0.1 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.

This program, *lookat.c,* illustrates how to use a viewing transformation with a projection transformation:

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

long v[8][3] = {
   {-1, -1, -1},
   {-1, -1, 1},
   {-1, 1, 1},
   {-1, 1, -1},
   { 1, -1, -1},
   { 1, -1, 1},
   { 1, 1, 1},
   { 1, 1, -1},
};
int path[16] = {
   0, 1, 2, 3,
   0, 4, 5, 6,
   7, 4, 5, 1,
   2, 6, 7, 3
};

void drawcube()
{
   int i;
   bgnline();
   for (i = 0; i < 16; i++)
      v3i(v[path[i]]);
   endline();
}

main()
{
   prefsize(600, 400);
   winopen("lookat");
   mmode(MVIEWING);
   perspective(300, 1.5, 0.1, 10.0);
   lookat(5.0, 4.0, 6.0, 1.0, 1.0, 1.0, 0);
   color(BLACK);
   clear();
   color(WHITE);
   drawcube();
   sleep(10);
   gexit();
   return 0;
}
```

7.4 Modeling Transformations

When you create a geometry, the GL 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()`. Each time you specify a transformation such as `rotate()` or `translate()`, the software automatically generates a *transformation matrix* that premultiplies the current matrix by the factors by which the coordinate system is to be rotated or translated. See Appendix C for the transformation matrices used by the GL.

Figure 7-8 shows some examples of modeling transformations.

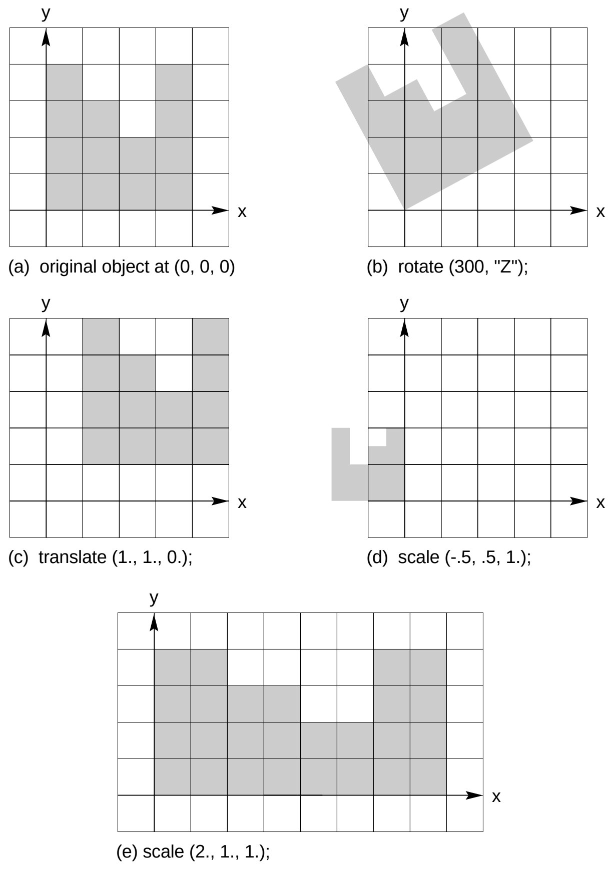


Figure 7-8 Modeling Transformations

7.4.1 Rotation

Use `rotate()` to rotate a geometry by specifying an *angle* and an *axis* of rotation:

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

The angle is given in tenths of degrees according to the right-hand rule. A character (either upper- or lowercase *x*, *y*, or *z*), defines the axis of rotation. The angle and axis are used to compute a 4x4 rotation matrix *R* (see Appendix C), that premultiplies the current matrix *T* to give *RT*.

`rot()` is the same as rotate; it specifies an angle and an axis of rotation in floating point values, but the angle is measured in degrees:

```
void rot(Angle a, char axis)
```

You need to pay close attention to the order in which you specify transformation operations, or your program might provide you with surprising results. See Section 7.5, "Controlling the Order of Transformations," and Figure 7-9 for more information about the importance of the order of modeling transformations and how to preserve untransformed coordinates.

The geometry in Figure 7-8 (a) is rotated 30 degrees with respect to the *y* axis in Figure 7-8 (b). All geometries drawn after you call `rotate()` or `rot()` are rotated. Use `pushmatrix()` and `popmatrix()` to preserve and restore the unrotated coordinate system.

7.4.2 Translation

Use `translate()` to move the coordinate system origin to a point (*x*,*y*,*z*) specified in the current coordinate system:

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

The *x*,*y*, and *z* coordinates are used to compute a 4×4 translation matrix *X* (see Appendix C), that premultiplies the current matrix *T*, to give *XT*.

The geometry in Figure 7-8 (a) is translated by (1,1,0) in Figure 7-8 (c).

All geometries drawn after you call `translate()` are translated. Use `pushmatrix()` and `popmatrix()` to preserve and restore the untranslated coordinate system.

7.4.3 Scaling

Use `scale()` to shrink, expand, or mirror a geometry:

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

The *x, y,* and *z* scale factors are used to compute a 4×4 scale matrix *S* (see Appendix C) that premultiplies the current matrix *T* to give *ST.* Values with magnitudes of more than 1 cause expansion; values with magnitudes of less than 1 cause shrinkage. Negative values cause mirroring.

Note: There may be a performance penalty for using non-uniform scaling if you are using lighting calculations.

The geometry in Figure 7-8 (a) is shrunk to one-quarter of its original size and is mirrored about the *y* axis in Figure 7-8 (d). It is scaled only in the *x* direction in Figure 7-8 (e). All geometries drawn after you call `scale()` are scaled. Use `pushmatrix()` and `popmatrix()` to preserve and restore the unscaled coordinate system.

You can combine `rotate()`, `rot()`, `translate()`, and `scale()` to produce more complicated transformations. The order in which you apply these transformations is important. Figure 7-9 shows two sequences of `translate()` and `rotate()`. Each sequence has different results.

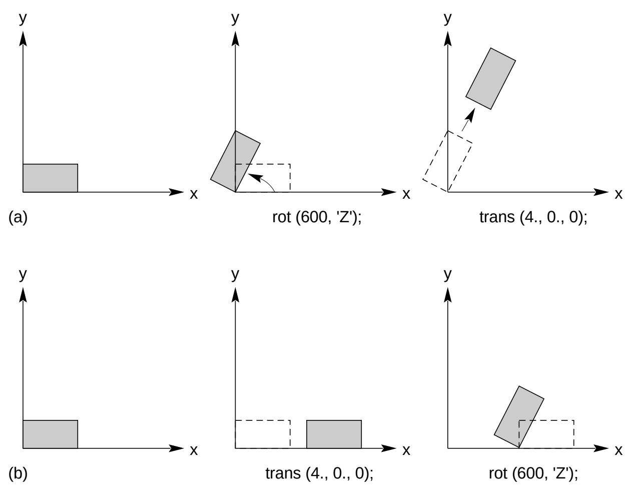


Figure 7-9 Effects of Sequence of Translations and Rotations

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 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.

All programs should set matrix mode to `MVIEWING`, `MPROJECTION`, or `MTEXTURE`, depending on which operation is to be done, before any matrix operations are performed. 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. For historical reasons, however, `MSINGLE` is the default.

Use `mmode()` to specify which of three matrices is the current matrix: ModelView (`MVIEWING`), Projection (`MPROJECTION`), or Texture (`MTEXTURE`):

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

The current matrix is on the top of the matrix stack.

When you are not doing lighting calculations, you should use MVIEWING for the modeling, viewing, and projection transformations. See Chapter 9 for information about the transformation matrices when lighting is used.

Note: Even in `mmode(MVIEWING)`, any calls to projection transformations `(perspective, window, ortho or ortho2)` will affect the Projection matrix.

7.6 Hierarchical Drawing with the Matrix Stack

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.

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, the current ModelView matrix sits at the top of the matrix stack; it contains all the modeling and viewing transformations called thus far. In the bicycle example, this lumped-together transformation is called *M*. Any vertex is transformed by the top matrix, which is 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. Translating by a translation matrix *T* leaves the stack with *TM* on top and *M* underneath. The wheel is then drawn once using the *SM* transformation. `popmatrix()` eliminates the *TM* on top, leaving *M*, and another `pushmatrix()` makes two copies of *M*.

```
... /* 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();
```

pushmatrix

Use `pushmatrix()` to push down the transformation stack, duplicating the current matrix:

```
void pushmatrix(void)
```

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

popmatrix

Use `popmatrix()` to pop the top matrix off the transformation stack:

```
void popmatrix(void)
```

Call `popmatrix()` only while `mmode()` is `MVIEWING`.

Note: It is important to have the same number of matrix pushes and matrix pops, so you don't try to pop a matrix off an empty stack.

This sample program, *hierarchy.c*, uses a hierarchical description of a simple car. It is so simple that the car body is a rectangle, 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 <math.h>
#include <gl/gl.h>

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

float carbody[4][XY] = {
   {-0.1, -0.05},
   { 0.1, -0.05},
   { 0.1, 0.05},
   {-0.1, 0.05}
};
float wheel[4][XY] = {
   {-0.015, -0.015},
   { 0.015, -0.015},
   { 0.015, 0.015},
   {-0.015, 0.015}
};

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

```
void drawcar()
{
   int i;
   color(RED);
   bgnpolygon();
      v2f(carbody[0]);
      v2f(carbody[1]);
      v2f(carbody[2]);
      v2f(carbody[3]);
   endpolygon();
   for (i = 0; i < 4; i++) {
      pushmatrix();
         translate(carbody[i][X], carbody[i][Y], 0.0);
         rotate(200*(i+1), 'z');
         drawwheel();
      popmatrix();
   }
}
main()
{
   float xoffset, yoffset;
   Angle ang;

   prefsize(400, 400);
   winopen("hierarchy");
   mmode(MVIEWING);
   ortho2(-1.0, 1.0, -1.0, 1.0);
   color(BLACK);
   clear();
   for (xoffset = -0.5; xoffset <= 0.5; xoffset += 0.5) {
      for (yoffset = -0.5; yoffset <= 0.5; yoffset += 0.5) {
         ang = 3600 * drand48();
         pushmatrix();
            translate(xoffset, yoffset, 0.0);
            rotate(ang, 'z');
            drawcar();
      popmatrix();
   }
 }
 sleep(10);
 gexit();
 return 0;
}
```

This shorter and perhaps more interesting program illustrates hierarchy and more complex transformations. It is a computer model of a ring and gear drawing toy. In the toy, a plastic ring is pinned to a piece of paper, and you place a pen through a hole in a smaller gear inside the ring and rotate the gear. As the moving gear rolls around the fixed gear, you draw interesting patterns.

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

#define PEN_TO_CENTER   0.2
#define R0              0.35
#define R1              0.6

void drawdot()
{
   translate(PEN_TO_CENTER, 0.0, 0.0);
   pnt2i(0, 0);
}

void 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;

   prefsize(400, 400);
   winopen("spirograph");
   mmode(MVIEWING);
   ortho2(-2.0, 2.0, -2.0, 2.0);
   color(BLACK);
   clear();
   color(WHITE);
   for (theta = 0.0; theta < 3600.0; theta += 0.25)
      draw1(theta);
   sleep(10);
   gexit();
   return 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 sophisticated drawing program 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_CENTER0 0.2
#define R0             0.35
#define R1             0.6
#define R2             0.8

void drawdot()
{
   translate(PEN_TO_CENTER, 0.0, 0.0);
   pnt2i(0, 0);
}

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

void 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;
 prefsize(400, 400);
 winopen("spirograph2");
 mmode(MVIEWING);
 ortho2(-3.0, 3.0, -3.0, 3.0);
 color(BLACK);
 clear();
 color(WHITE);
 for (theta = 0.0; theta < 18000.0; theta += 0.25)
   drawx(theta);
 sleep(10);
 gexit();
 return 0;
}
```

7.7 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 visible screen area varies from system to system.

viewport

Use `viewport()` to specify, in window coordinates, the area of the window that displays an image:

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

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. By default, when you open a window on the screen, its viewport is set to cover the whole window.

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)

Note: To correctly map object coordinates one-to-one to window coordinates, call:

```
ortho(-0.5, width -0.5, -0.5, height-0.5)
viewport(0, width-1, 0, height-1);
```

where *width* and *height* are the integer pixel sizes of the window.

getviewport

Use `getviewport()` to return the current viewport:

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

The 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.

scrmask

The screenmask is a specified rectangular area of the screen to which all drawings are clipped. Use `scrmask()` to define the screenmask:

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

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. `viewport()` sets both the viewport and the screenmask to the same area; `scrmask()` sets only the screenmask, which must be placed entirely within the viewport. See Chapter 3 for one use of a screenmask—clipping characters.

getscrmask

Use `getscrmask()` to return the coordinates of the current screenmask in the arguments *left*, *right*, *bottom*, and *top*:

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

pushviewport

The system maintains a stack of viewports, and the top element in the stack is the current viewport. Use `pushviewport()` to duplicate the current viewport and screenmask and push them onto the stack:

```
void pushviewport(void)
```

popviewport

Use `popviewport()` to pop the stack of viewports and set the viewport and screenmask (the viewport on top of the stack is lost):

```
void popviewport(void)
```

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. Call scrbox with the desired mode:

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

While enabled (mode `SB_TRACK`), `scrbox()` maintains left-most, right-most, lowest, and highest window coordinates of all pixels that are scan-converted. By default, `scrbox()` is reset (mode `SB_RESET`), forcing the left-most and lowest screen box values to be greater than the right-most 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.

getscrbox

Use `getscrbox()` to return the current scrbox into *left, right, bottom,* and *top*:

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

7.8 User-Defined Transformations

Modeling and viewing transformation commands premultiply the current matrix (one of ModelView, Projection, or Texture) with a 4×4 matrix that they compute based on their parameters.

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`.

You can also premultiply, or replace, the current matrix with a 4x4 matrix of your own.

Use `multmatrix()` to premultiply the current matrix by the given matrix (*m*):

```
void multmatrix (Matrix m)
```

That is, if *T* is the current matrix, `multmatrix(m)` replaces *T* with *MT*.

Use `getmatrix()` to copy the current matrix to an array:

```
void getmatrix(Matrix m)
```

Use `loadmatrix()` to load a 4×4 floating point matrix, *m*, onto the stack, replacing the current top of the stack:

```
void loadmatrix(Matrix m)
```

7.9 Additional Clipping Planes

Geometry is always clipped against the boundaries of the six-plane viewing volume defined by the current Projection matrix. `clipplane()` allows the specification of additional planes, not necessarily perpendicular to the *x*, *y*, or *z* axis, 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(long index,long mode,float params[])
```

where:

index	integer in the range 0 through 5. Indicates which of the six clipping planes is being modified.
mode	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	array of four floats that specify a plane equation. A plane equation is usually thought of as the column 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()` in Chapter 2) can be passed as a plane equation, where vertex *x* becomes *A*, *y* becomes *B*, and so on.

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

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}, as shown in (EQ 7-1).

(EQ 7-1)

$$P_{\text{object}} = \begin{bmatrix} A \\ B \\ C \\ D \end{bmatrix}$$

$$P_{\text{eye}} = M^{-1}{}_{\text{modelview}} \cdot P_{\text{object}}$$

Eye coordinates whose dot product with P_{eye} is positive or zero are inside the region, 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 enabled but undefined clipping plane(s) is also undefined.

It is sometimes convenient to define a clipping plane based on a point, and a direction in object coordinates. The plane equation is obtained from the dot product of the point and the normal:

(EQ 7-2)

$$\text{Point} = \begin{bmatrix} P_x & P_y & P_z \end{bmatrix}$$

$$\text{Normal} = \begin{bmatrix} N_x \\ N_y \\ N_z \end{bmatrix}$$

$$\text{Plane} = \begin{bmatrix} N_x \\ N_y \\ N_z \\ -\begin{bmatrix} P_x & P_y & P_z \end{bmatrix} \bullet \begin{bmatrix} N_x \\ N_y \\ N_z \end{bmatrix} \end{bmatrix} = \begin{bmatrix} A \\ B \\ C \\ D \end{bmatrix}$$

Chapter 8

Hidden-Surface Removal

When you look at a 3-D scene, you see only the surfaces that are nearest to the eye, while other surfaces behind them are obscured (provided the items are opaque). Drawing speed can be improved by drawing only the items that are visible to the eye in the final scene. *Hidden-surface removal* is the process of determining in advance which surfaces are not visible in the final scene, in order to prevent them from being drawn. This chapter describes how to do hidden-surface removal.

- Section 8.1, "z-buffering," tells you how to remove hidden surfaces by drawing only the surface closest to the eye.
- Section 8.2, "Using z-buffer Features for Special Applications," tells you how to use other techniques for determining visible surfaces.
- Section 8.3, "Stenciling," tell you how to create stencils that allow you to update drawings selectively.
- Section 8.4, "Eliminating Backfacing Polygons," tells you how to remove hidden surfaces by drawing only the polygons that face the viewer.
- Section 8.5, "Alpha Comparison," tells you how to use special alpha hardware to indicate transparency.

Two basic methods of hidden-surface removal are discussed in this chapter. One method of determining surface visibility is to use a *z*-buffer to keep track of which item is closest to the eye at each pixel. Another method is to disable drawing for polygons that face away from the viewer. Backfacing polygon removal is not as general as *z*-buffering, but *z*-buffering may be slower than backface removal on some systems.

8.1 *z*-buffering

The *z-buffer* is a bitplane, associated with a framebuffer, that stores the distance from the near clipping plane to each pixel in the window. In *z*-buffer mode, the *z* coordinate (distance to the eye) of the incoming (next to be drawn) 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 existing geometry, the values of the old pixel and of the old *z* value, which are stored in the color framebuffer and the *z*-buffer, are replaced by the new ones.

The calculation is performed 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, as shown in Figure 8-1.

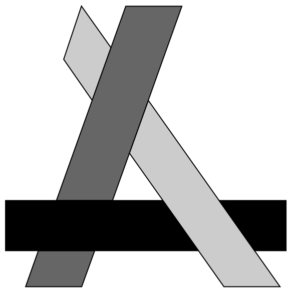


Figure 8-1 Overlapping Polygons

Not all systems support z-buffering. Use `getgdesc(GD_BITS_NORM_ZBUFFER)` to return the z-buffer availability. The draw mode must be set to `NORMALDRAW` to use z-buffering.

On most systems, the *z*-buffer is a hardware option. The size of the *z*-buffer can be from 24 bits to 32 bits per screen pixel, depending on the system type. The *z* value is signed on all systems except IRIS-4D/GT/GTX systems.

RealityEngine systems provide software support for selecting the size of the hardware z-buffer to be 0 or 32 bits and support for multisampled z-buffering, which is described in Chapter 15. See *zbsize*(3G) for more information.

The IRIS Indigo Entry system has a software *z*-buffer, with 32 bit *z*-buffer memory allocated on a per window basis, rather than per screen.

A 24-bit hardware *z*-buffer is optional on XS and XS24 systems and is standard on Elan systems. On Elan systems, the low bit of the 24-bit *z* buffer is reserved for fast clears, so that bit should be ignored when reading data back from the *z*-buffer. Elan systems also allocate bits from the *z*-buffer for stencil operations, so the resolution of the *z*-buffer is decreased when performing stenciling.

By default, *z*-buffering is turned off. To set up *z*-buffering, you enable *z*-buffer mode, then write the maximum *z* value to every location in the *z*-buffer, using the following commands:

```
zbuffer(TRUE);/* Enable z-buffering */
zclear();     /* Write the maximum z value to the z-buffer */
```

Before the system draws anything, it compares the *z* value of each incoming pixel to the *z*-buffer value for that pixel. If the *z* value of the incoming pixel is smaller than the value in the *z*-buffer, the pixel is colored, and that pixel's *z*-buffer value is set to the new *z* value. If the incoming pixel's *z* value is greater than the corresponding *z*-buffer value, the pixel is not drawn. The values in the *z*-buffer thus always 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. Use `getzbuffer()` to determine whether or not *z*-buffering is enabled; TRUE means *z*-buffering is enabled and FALSE means it is not enabled.

Another consideration when using *z*-buffering is that the *znear* and *zfar* values in the call to `perspective()` have a profound effect on the resolution of the *z*-buffer's comparison facility. The *z*-buffer contains a finite number of integers, each with a limited range of values that can be used to compare against the *z* value of the incoming pixel. You can enhance the resolution of the *z*-buffer by setting the near and far values as close together as possible. With a smaller range of total values, more bits of precision are available. It is particularly important to move the near clipping plane as far from the eye as possible.

This sample program, *zbuffer.c*, draws three rectangular boxes that tumble through one another while the whole scene rotates. While the left mouse button is up, the scene is drawn without *z*-buffering; when you press it, *z*-buffering is enabled. If you run the program with an argument—by entering **`zbuffer 5`**, for example, there is a short delay before drawing each polygon. The left mouse button still controls the *z*-buffering, but it is easier to see what is happening because you can see each polygon drawn one at a time.

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

float v[8][3] = {
    {-1.0, -1.0, -1.0},
    {-1.0, -1.0, 1.0},
    {-1.0, 1.0, 1.0},
    {-1.0, 1.0, -1.0},
    { 1.0, -1.0, -1.0},
    { 1.0, -1.0, 1.0},
    { 1.0, 1.0, 1.0},
    { 1.0, 1.0, -1.0},
};
unsigned int delaycount;

void delay()
{
    if (delaycount)
        sleep(delaycount);
}
void drawcube()
{
    color(RED);
    bgnpolygon();
        v3f(v[0]);
        v3f(v[1]);
        v3f(v[2]);
        v3f(v[3]);
    endpolygon();
    delay();
    color(GREEN);
    bgnpolygon();
        v3f(v[0]);
        v3f(v[4]);
        v3f(v[5]);
        v3f(v[1]);
    endpolygon();
    delay();
    color(BLUE);
    bgnpolygon();
        v3f(v[4]);
        v3f(v[7]);
        v3f(v[6]);
        v3f(v[5]);
    endpolygon();
    delay();
    color(YELLOW);
    bgnpolygon();
        v3f(v[3]);
        v3f(v[7]);
        v3f(v[6]);
        v3f(v[2]);
    endpolygon();
    delay();
    color(MAGENTA);
    bgnpolygon();
        v3f(v[5]);
        v3f(v[1]);
        v3f(v[2]);
        v3f(v[6]);
    endpolygon();
    delay();
    color(CYAN);
    bgnpolygon();
        v3f(v[0]);
        v3f(v[4]);
        v3f(v[7]);
        v3f(v[3]);
    endpolygon();
}

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

    xrot = yrot = zrot = 0;
    if (argc == 1)
        delaycount = 0;
    else
        delaycount = 1;
```

```
if (getgdesc(GD_BITS_NORM_ZBUFFER) == 0) {
    fprintf(stderr, "Z-buffer not available on this machine\n");
    return 1;
}
prefsize(400, 400);
winopen("zbuffer");
if (delaycount == 0)
    doublebuffer();
gconfig();
mmode(MVIEWING);
ortho(-4.0, 4.0, -4.0, 4.0, -4.0, 4.0);
qdevice(ESCKEY);
qdevice(LEFTMOUSE);  /* don't want window manager to act on clicks */

while (!(qtest() && qread(&val) == ESCKEY && val == 0)) {
    pushmatrix();
        rotate(xrot, 'x');
        rotate(yrot, 'y');
        rotate(zrot, 'z');
        color(BLACK);
        clear();
        if (getbutton(LEFTMOUSE)) {
            zbuffer(TRUE);
            zclear();
        }
        else
            zbuffer(FALSE);
        pushmatrix();
            scale(1.2, 1.2, 1.2);
            translate(0.3, 0.2, 0.2);
            drawcube();
        popmatrix();
        pushmatrix();
            rotate(450 + zrot, 'x');
            rotate(300 - xrot, 'y');
            scale(1.8, 0.8, 0.8);
            drawcube();
        popmatrix();
        pushmatrix();
            rotate(500 + yrot, 'z');
            rotate(-zrot, 'x');
            translate(-0.3, -0.2, 0.6);
            scale(1.4, 1.2, 0.7);
            drawcube();
        popmatrix();
    popmatrix();
```

```
        if (delaycount == 0)
            swapbuffers();
        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;
    }
    gexit();
    return 0;
}
```

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

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

First, `zbuffer(TRUE)` enables *z*-buffer comparisons to be made before each write, then `zclear()` 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 normally 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 is turned off.

8.1.1 Controlling *z* Values

Just as `viewport()` controls the scaling of *x* and *y* coordinates, there is a subroutine, `lsetdepth()`, that controls the scaling of *z* coordinates. `lsetdepth()` takes two arguments, corresponding to the near and far clipping 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. Use `getgdesc(GD_ZMIN)` to return the minimum *z* value and `getgdesc(GD_ZMAX)` to return the maximum *z* value, so that you can set *near* and *far* to their minimum and maximum values, regardless of what type of system is used, by calling:

```
lsetdepth(getgdesc(GD_ZMIN), getgdesc(GD_ZMAX));
```

Table 8-1 lists the minimum and maximum *z* values for different models. These are signed values on all systems except IRIS-4D/GT/GTX systems.

System Model	Minimum Value	Maximum Value
IRIS-4D/B or G	0x4000	0x3FFF
IRIS-4D/GT or GTX	0	0x7FFFFF
Personal IRIS, IRIS-4D/VGX, VGXT, SkyWriter, XS, XS24, Elan	0x800000	0x7FFFFF
IRIS Indigo, RealityEngine	0x80000000	0x7FFFFFFF

Table 8-1 Maximum and Minimum *z*-buffer Values

Note: *z*-buffer hardware is optional on the XS, XS24, and Elan. These systems do not provide a software *z*-buffer in lieu of the hardware *z*-buffer.

8.1.2 Clearing the *z*-buffer and the Bitplanes Simultaneously

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

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

This code clears the color bitplanes to zero, then clears the *z*-buffer bitplanes to the maximum value. Unfortunately, it takes a relatively long time, because `clear()` touches each pixel first, then `zclear()` touches each pixel again. In recent hardware implementations, the hardware can, in certain cases, simultaneously clear the color planes and the *z*-buffer planes. Use `czclear()` to do simultaneous clearing of color and *z*-buffer:

```
czclear(long color, long zval)
```

The circumstances and results of using `czclear()` are different on different systems. See the *czclear(3G)* man page for details. `czclear()` clears the bitplanes to *color* and the *z*-buffer to *zval* simultaneously.

Using czclear on IRIS-4D/VGX, VGXT and SkyWriter Systems

IRIS-4D/VGX, VGXT, SkyWriter, and RealityEngine always clear the banks of color and *z* bitplanes sequentially, regardless of the values of *cval* and *zval*.

Using czclear on IRIS-4D/GT/GTX Systems

On IRIS-4D/GT/GTX systems, `czclear()` simultaneously clears the *z*-buffer and bitplanes if circumstances allow it. These systems can perform a simultaneous clear under the following conditions:

- 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 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 all *z* values are initially cleared is 0 instead of ZMAX:

  ```
  lsetdepth(getgdesc(GD_ZMAX), 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 an index into the color map index, only the lowest 12 bits are significant.

Using czclear on Personal IRIS, XS, XS24, and Elan Systems

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. See Section 8.2.2, "Alternative Comparisons and z-buffer Writemasks,"for information about `zfunction()`.

zval	zfunction
getgdesc(`GD_ZMIN`)	`ZF_GREATER` or `ZF_GEQUAL`
getgdesc(`GD_ZMAX`)	`ZF_LESS` or `ZF_LEQUAL`

Table 8-2 Values of `zfunction()` for Personal IRIS `czclear()`

8.2 Using z-buffer Features for Special Applications

This section discusses special features associated with *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.

8.2.1 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 writing into the *z*-buffer, where the values in the *z*-buffer are all set to a particular depth value.

In a flight simulator, for example, suppose that the view on the screen includes an instrument panel surrounding the plane's windshield. 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 view out the plane's windshield and redraw only that portion of the window for each frame.

To do this, set the current "color" to the value returned by the call to `getgdesc(GD_ZMAX)` on your system. Use `zdraw()` to write this value into the *z*-buffer, then draw the polygon(s) representing the windshield. When the outside view is drawn, it is always masked by the plane's windshield frame and instrument panel (which is closer to the eye). Thus an extremely complex instrument panel is possible, because it needs to be drawn only once.

When `zdraw()` is TRUE, `zbuffer()` must be set to FALSE, otherwise both try to alter the *z*-buffer contents simultaneously. In color map mode, the value written to the *z*-buffer by `zdraw()` is the index of the color specified. For example, `color(2)` writes 2s to the *z*-buffer.

`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. Usually, drawing into the *z*-buffer should be bracketed by subroutines that first set `backbuffer(FALSE)` and then `backbuffer(TRUE)`, assuming that 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 (which is the only buffer in single buffer mode) is not written. If `zdraw()` is FALSE, `frontbuffer(FALSE)` has no effect.

On Elan, XS, and XS24 systems, do not use `zdraw()` when drawing into a clipped window, or when using read/modify/write operations such as antialiasing or `logicop()`.

This sample program, *zdraw.c*, illustrates the `zdraw()` masking technique. It draws a spinning cube, seen through square cut-outs in a green wall.

```
#include <stdio.h>
#include <gl/gl.h>
#include <gl/device.h>
#define RGB_BLACK 0x000000
#define RGB_GREEN 0x00ff00
#define HOLESIZE 32
#define HOLESEP (HOLESIZE/2)

float v[8][3] = {
    {-1.0, -1.0, -1.0},
    {-1.0, -1.0, 1.0},
    {-1.0, 1.0, 1.0},
    {-1.0, 1.0, -1.0},
    { 1.0, -1.0, -1.0},
    { 1.0, -1.0, 1.0},
    { 1.0, 1.0, 1.0},
    { 1.0, 1.0, -1.0},
};

int face[6][4] = {
    {0, 1, 2, 3},
    {3, 2, 6, 7},
    {7, 6, 5, 4},
    {4, 5, 1, 0},
    {1, 2, 6, 5},
    {0, 4, 7, 3},
};

unsigned long facecolor[6] = {
    0xff0000,    /* blue */
    0x0000ff,    /* red */
    0x00ffff,    /* yellow */
    0xffff00,    /* cyan */
    0xff00ff,    /* magenta */
    0xffffff,    /* white */
};
```

```
void drawcube()
{
    int i;
    for (i = 0; i < 6; i++) {
        cpack(facecolor[i]);
        bgnpolygon();
            v3f(v[face[i][0]]);
            v3f(v[face[i][1]]);
            v3f(v[face[i][2]]);
            v3f(v[face[i][3]]);
        endpolygon();
    }
}
main(argc, argv)
int argc;
char *argv[];
{
    int i, j;
    Angle xang, yang;
    short val;
    Boolean use_geom;
    unsigned long holez[HOLESIZE*HOLESIZE];

    if (getgdesc(GD_BITS_NORM_DBL_RED) == 0) {
        fprintf(stderr, "Double buffered RGB not available on this machine\n");
    return 1;
    }
    if (getgdesc(GD_BITS_NORM_ZBUFFER) == 0) {
        fprintf(stderr, "Z-buffer not available on this machine\n");
        return 1;
    }
    if (getgdesc(GD_ZDRAW_GEOM) == 0 && getgdesc(GD_ZDRAW_PIXELS) == 0) {
        fprintf(stderr, "Z-buffer drawing not available on this machine\n");
        return 1;
    }
    prefsize(400, 400);
    winopen("zdraw");
    RGBmode();
    doublebuffer();
    gconfig();
    qdevice(ESCKEY);
    mmode(MVIEWING);
    ortho(-2.0, 2.0, -2.0, 2.0, -2.0, 2.0);
    zbuffer(TRUE);
    zclear();
    use_geom = getgdesc(GD_ZDRAW_GEOM) == 1;
```

```
if (!use_geom) {
    holez[0] = getgdesc(GD_ZMAX);
    for (i = 1; i < HOLESIZE*HOLESIZE; i++)
        holez[i] = holez[0];
}

/* draw the green wall once */
cpack(RGB_GREEN);
frontbuffer(TRUE);
pushmatrix();
    translate(0.0, 0.0, 1.9);
    rectf(-2.00, -2.00, 2.00, 2.00);
popmatrix();
frontbuffer(FALSE);
xang = yang = 0;
while (!(qtest() && qread(&val) == ESCKEY && val == 0)) {
/* create the holes in the green wall */
zbuffer(FALSE);
zdraw(TRUE);
backbuffer(FALSE);
if (use_geom) {
    ortho2(-0.5, 399.5, -0.5, 399.5);
    cpack(getgdesc(GD_ZMAX));
}
for (i = 100; i <= 300; i += 50) {
    for (j = 100; j <= 300; j += 50) {
    if (use_geom)
        rectf(i, j, i + HOLESIZE - 1, j + HOLESIZE - 1);
    else
        lrectwrite(i, j, i +HOLESIZE - 1, j + HOLESIZE - 1, holez);
    }
}
if (use_geom)
    ortho(-2.0, 2.0, -2.0, 2.0, -2.0, 2.0);
zdraw(FALSE);
backbuffer(TRUE);
zbuffer(TRUE);

/* z-buffered clear to background color and depth */
cpack(RGB_BLACK);
pushmatrix();
    translate(0.0, 0.0, -1.9);
    rectf(-2.00, -2.00, 2.00, 2.00);
popmatrix();
```

```
    /* draw the outside scene */
    pushmatrix();
        rotate(xang, 'x');
        rotate(yang, 'y');
        drawcube();
    popmatrix();
    swapbuffers();

    /* update the rotation angles for next time through */
    xang += 11;
    yang += 17;
    if (xang > 3600)
        xang -= 3600;
    if (yang > 3600)
        yang -= 3600;
    }
    gexit();
    return 0;
}
```

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.

The idea behind the program is that a green wall is drawn nearer the eye than anything else. This sets all the *z*-buffer values so they record data near the eye. In the main loop, 25 holes are drilled into the wall by setting the *z*-buffer values in the squares 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, and it is visible only through the holes.

8.2.2 Alternative Comparisons and *z*-buffer Writemasks

In the default *z*-buffer 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 another type of decision, to achieve a different effect. To change the comparison function, use `zfunction()`.

The *z* comparison functions are:

`ZF_NEVER`	Never overwrite the source pixel value.
`ZF_LESS`	Overwrite the source pixel value if the *z* value of the source pixel is less than the *z* value of the destination pixel.
`ZF_EQUAL`	Overwrite the source pixel value if the *z* value of the source pixel is equal to the *z* value of the destination pixel.
`ZF_LEQUAL`	Overwrite the source pixel value if the *z* value of the source pixel is less than or equal to the *z* value of the destination pixel (default).
`ZF_GREATER`	Overwrite the source pixel value if the *z* value of the source pixel is greater than the *z* value of the destination pixel.
`ZF_NOTEQUAL`	Overwrite the source pixel value if the *z* value of the source pixel is not equal to the *z* value of the destination pixel.
`ZF_GEQUAL`	Overwrite the source pixel value if the *z* value of the source pixel is greater than or equal to the *z* value of the destination pixel.
`ZF_ALWAYS`	Always overwrite the source pixel value regardless of value of the destination pixel.

You can also control writing into the *z*-buffer with `zwritemask()`. This might be useful for using a very complicated background into which a few items are to be drawn and moved quickly. Setting `zwritemask()` to zero locks the background information in, and prevents its modification; the new items are drawn or not depending on the depth comparison.

8.3 Stenciling

IRIS-4D/VGX, SkyWriter, and RealityEngine 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 flexible frame buffer configuration on these systems to allocate *stencil planes*.

Stenciling, like *z*-buffering, enables and disables drawing on a per-pixel basis. You draw into the stencil planes using GL drawing primitives, then render geometry and images, using the stencil planes to mask out portions of the screen. Stenciling is typically used in multipass rendering algorithms to achieve special effects, like decals, outlining, and constructive solid geometry rendering.

The `stencil()` statement controls testing of stencil bitplanes before the system writes to the frame buffer:

```
void stencil(long enable, unsigned long ref, long func,
          unsigned long mask, long fail, long pass, long zpass);
```

The first argument to `stencil()` enables or disables stenciling. 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.

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 for the *z*-buffer) by a programmed value, as defined in the call to `stencil()`.

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 *fail*, *pass*, and *zpass*) regardless of the results of the comparison.
`SF_LESS`	Perform the specified stencil update (specified as the value of *fail, pass,* 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 *fail, pass,* 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 *fail, pass,* 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 *fail, pass,* 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 *fail, pass,* 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 *fail, pass,* 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 *fail, pass,* 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.

These three possible values are reflected as the arguments *fail*, *pass*, and *zpass* (the last three arguments passed to stencil). If the *z*-buffer is not enabled, only *fail* and *pass* are considered. 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* passed to `stencil()`.

`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.

A sample stencil command follows, with a description of its results.

```
stencil(TRUE, 220, SF_EQUAL, 0xff, ST_REPLACE, ST_KEEP, ST_KEEP);
```

In this example, 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.

stensize

Use `stensize()` to define the bitplanes you wish to use as the stencil:

```
void stensize(long planes)
```

You can define up to 8 stencil planes. 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 a number of bitplanes for use as a stencil, these planes can be used to store information that is later used by the `stencil()` statement.

sclear

Use `sclear()` to set the value of every pixel in the stencil buffer:

```
void sclear(unsigned long sval)
```

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()`.

swritemask

Use `swritemask()` to specify which of the stencil bitplanes can be modified by `sclear()` and normal stencil operation:

```
void swritemask(unsigned long mask)
```

The next sample program, *stencil.c*, uses stenciling to render the outline of an object in an arbitrary image. Because the bitmap of an image takes a lot of space, the code mimics drawing the image by actually drawing polygons.

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.

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

floatrect0[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();
}
```

8.4 Eliminating Backfacing Polygons

In a scene composed entirely of opaque closed surfaces, *backfacing polygons* (polygons whose face is pointing away from the viewer) 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. This is especially useful on systems such as the IRIS Indigo that have slower *z*-buffer rates.

The idea is that if the polygons making up a surface are all oriented the same way, and if the surface is closed, after transformation, all the polygons on the front have one orientation and those on the back have the opposite orientation. A special mode can be turned on to check whether the transformed polygons are oriented clockwise or counterclockwise, and only those oriented counterclockwise are drawn. The method is not sufficient for all hidden surface removal if the object being drawn is not convex, or if there is more than one object.

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 such that they are drawn in counterclockwise order when the front face of the polygon is visible (see Chapter 2 for examples).

Use `backface()` to initiate or terminate 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 side of the solid facing away from the viewer are in clockwise order and are not drawn. `backface()` takes a single argument. TRUE enables backfacing polygon elimination, and FALSE (the default) disables it.

Use `frontface()` to initiate or terminate frontfacing polygon removal. The `frontface()` utility is used to display hidden surfaces (backfacing polygons). In this case, the polygons on the side of the solid facing away from the viewer are drawn; those facing the viewer are not drawn. `frontface()` takes a single argument. TRUE enables frontfacing polygon elimination, and FALSE (the default) disables it.

`getbackface()` returns the state of backfacing 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.

8.5 Alpha Comparison

On some systems, you can also use the alpha planes to determine whether to draw pixels by comparing incoming alpha values to a reference constant value. Not all systems support alpha operations. Use `getgdesc(GD_BITS_NORM_SNG_ALPHA)` to test for the availability of alpha planes.

Note: RealityEngine systems feature multisampled alpha comparison, which is described in Chapter 15.

Use `afunction()` to compare the alpha values of source pixels to the value of *ref*:

```
void afunction(long ref, long func)
```

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()`, `linesmooth()`, or `polysmooth()`.

The `afunction()` 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()`, `linesmooth()`, or `polysmooth()`.

The `afunction()` statement compares source alpha values against a reference value that you include in the `afunction()` call. You also specify a comparison function that determines the conditions under which `afunction()` permits the system to draw pixels.

To make the system avoid drawing invisible pixels, call `afunction()` as follows:

```
afunction(0, AF_NOTEQUAL);
```

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 sample program, *afunction.c,* uses `afunction()` to define the shape of a building and its windows. See Chapter 18 to learn how to use texturing.

```
#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;
}
```

Chapter 9

Lighting

This chapter describes the GL lighting facility and tells you how to create lighted scenes.

- Section 9.1, “Introduction to GL Lighting,” presents some terminology and basic principles of lighting that you need to know to understand GL lighting.
- Section 9.2, “Setting Up GL Lighting,” tells you how to configure lighting parameters.
- Section 9.3, “Binding Lighting Definitions,” tells you how to activate your lighting definitions.
- Section 9.4, “Changing Lighting Settings,” tells you how to change lighting definitions that you have set up.
- Section 9.5, “Default Settings,” describes the default lighting settings and tells you how to create your own default definitions.
- Section 9.6, “Advanced Lighting Features,” tells you how to use special lighting features for more complex lighting effects.
- Section 9.7, “Lighting Performance,” gives you some programming hints to help you get the best performance from lighting.
- Section 9.8, “Color Map Lighting,” tells you how to use lighting in color map mode, primarily for machines with limited RGB capability.

Code fragments are used extensively to encourage a learn-by-example approach. A complete sample program is included at the end of the chapter.

9.1 Introduction to GL Lighting

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 that strikes the object from the light source is called *incident* light; the light that bounces off the object's surface is called *reflected* light.

In GL lighting, as in the real world, the characteristics of the light source determine the direction, intensity, and wavelength(color) of the incident light. The characteristics of the object geometry and its surface material determine the direction, intensity, and color of the reflected light. How light is reflected depends on the interaction between the incident light and the surface material.

The interaction between light and matter is far more complicated than can be simulated in real time. Lighting on the IRIS achieves a balance between realistic appearance and real-time drawing speed. The GL achieves this balance by performing lighting calculations only at geometry vertices, rather than calculating lighting for each pixel rendered.

You control GL lighting by creating a lighting definition that allows you to specify and manipulate the characteristics of the incident light, the surface properties of lighted objects, and the effects of the surrounding environment. In a lighting definition, you set up the color, intensity, and position of light sources, the properties of surface materials, and the characteristics of the lighting environment in your scene. Once you have your lighting definition set up, you can turn lights on and off, use different object materials, and use advanced lighting techniques for interesting effects.

9.1.1 Color

A lighting definition determines how the color of incident light is modified when it reflects from surfaces. For example, an object can appear blue because:

- A white light source shines on an object which 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.

GL lighting allows both of these methods: 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.

9.1.2 Reflectance

The ratio of incident light to reflected light is called *reflectance*. Some of the factors that determine reflectance are inherent in an object's geometry; other properties that affect reflectance are controlled by GL lighting.

There are three reflectance properties that determine how a surface reflects light:

- *diffuse*, which shows up as a matte, or flat, reflection
- *specular*, which shows up as highlights
- *ambient*, which simulates indirect light

Diffuse Reflectance

Diffuse reflectance gives the appearance of a matte, or flat, reflection from a surface. The direction of the light as it falls on the surface determines how bright the diffuse reflection is. Diffuse reflection is brightest where the incident light strikes the perpendicular to the surface.

For example, consider a distant light shining directly on the north pole of a sphere representing the earth. 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 viewer's perception of diffuse intensity is not affected by the viewing angle.

Specular Reflectance

Specular reflectance creates highlights and is dependent on the position of the viewpoint. For example, consider the glare in your rear view 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.

Ambient Reflectance

Diffuse and specular reflectance simulate how objects in the scene reflect light that comes directly from a light source. Ambient reflectance 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 in areas of the scene that receive no direct illumination.

Emission

Emission is the quality of giving off light. Adding an emission component is useful for simulating the appearance of lights in a scene. A GL object that emits light does not also automatically act as a light source; that is, it does not add illumination to a scene. For this object to act as a light source in the scene, you must add a light to the lighting model and position this light at the same location as the object that is emitting light.

9.2 Setting Up GL Lighting

This section introduces fundamental GL lighting concepts, and tells you how to create a static lighting environment.

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.

To set up GL lighting:

1. Define the properties for each material, light and lighting model that you want in your scene.
2. Activate (*bind*) your definitions.
3. Draw the scene. Provide surface normals, as discussed in Section 9.2.1, “Defining Surface Normals,” for all primitives to be lighted.

9.2.1 Defining Surface Normals

Surface normals are unit-length vectors that are perpendicular to a given surface at a particular vertex. They serve as input to the lighting formula, and are required for GL lighting.

When using GL NURBS surfaces (see Chapter 14, “Curves and Surfaces”), normals can be generated automatically from the surface descriptions.

Specify a surface normal for each vertex using the `n3f()` subroutine. The GL maintains a *current normal*, which remains the same until you change it. The current normal is analogous to the current color.

This example specifies 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();
```

This example specifies 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 vertices and the direction of the normals is significant. When the order of the vertices of the projected triangle is counterclockwise as seen from the viewer’s perspective, the front face of the triangle is the side toward the viewer. In this case, the normals of the triangle also point toward the viewer. This convention obeys the right-hand rule. The example triangle given above displays this behavior. The right-hand rule makes it possible to distinguish between the front and the back faces of the triangle. Although it is not necessary for you to follow the right-hand-rule when using lighting, doing so allows you to use backface elimination (see Chapter 8). Two-sided lighting, described in Section 9.6.3, requires you to follow the right-hand-rule.

Non-Unit-Length Normals

The previous section stated that normals must be unit-length, which is the default. The GL can handle non-unit-length normals, but there may be a performance penalty associated with their use. See Section 9.7, "Lighting Performance," for information about performance considerations with non-unit-length normals.

Use the `nmode()` subroutine to handle non-unit-length normals. The default mode is `NAUTO`, which automatically corrects for situations where normals you would expect to be unit-length become non-unit-length. When you know your normals are unit-length, use this mode:

```
nmode(NAUTO);
```

To configure the GL to automatically normalize (correct to unit length) *all* normals (when you are intentionally using non-unit length normals), use:

```
nmode(NNORMALIZE);
```

Note: Lighting does not have to be on to set `nmode()`.The `nmode()` command is not available on all systems, see the `nmode()` man page for details.

9.2.2 Defining Lighting Components

You configure three lighting components to define GL lighting for your scene:

- material - determines how a surface responds to illumination
- light source - determines the characteristics of the incident light
- lighting model - determines the behavior of the lighting environment

The combination of these three components determines the appearance, or more specifically, the color at each lighted vertex. The GL calculates the color at each lighted vertex by summing the total ambient light (scaled by the material ambient reflectance), the material emitted light, and the contributions of each light source.

The total ambient light is the sum of the ambient light associated with each light source and the ambient light associated with the scene, as given by the lighting model.

The contribution of each light source is the sum of:

1. Light source ambient color, scaled by material ambient reflectance.
2. Light source color, scaled by material diffuse reflection and the *dot product* (see Foley and Van Dam, *Computer Graphics Principles and Practice*) of the vertex normal and the vertex-to-light source vector.
3. Light source color, scaled by material specular reflectance and a function of the angle between the vertex to viewpoint vector and the vertex-to-light source vector.

You configure each of the three components using a two-step process. The first step is *definition*, using `lmdef()`. The second step is *activation*, using `lmbind()`.

Use the `lmdef()` subroutine to establish your material, light source and lighting model definitions.

The ANSI C specification of `lmdef()` is:

```
void lmdef(short deftype, short index, short np, float props[])
```

where:

deftype	indicates whether the properties in the array apply to a material, a light source, or a lighting model.
index	defines an index to be associated with the definition.
np	provides a count of symbols and floating point values in the props array. You can usually set np to 0, in which case it is ignored; however, operation over network connections is more efficient when np is correctly specified.
props	array of floating point symbols that define properties.

You specify properties as an array of floats. The array elements are terminated by the special constant `LMNULL`, which must always be the last float in the array. `lmdef()` associates the properties in your property array with an integer index and copies these properties at the time of the call.

Index 0 is reserved as the null definition for material, light source and lighting model definitions. You use index 0 to turn definitions off, as described in Section 9.4, “Changing Lighting Settings.” You cannot assign your own definition to index 0.

Defining a Material

To define a material, you specify its properties in an array of floats.

For example, a greenish plastic-like material is defined like this:

```
static float mat[] = {
   AMBIENT, .1, .1, .1,
   DIFFUSE, 0, .369, .165,
   SPECULAR, .5, .5, .5,
   SHININESS, 30,
   LMNULL
};
lmdef(DEFMATERIAL, 39, 5, mat);
```

In this example, the `lmdef()` call stores the material properties in the *mat* array as material number `39`. This creates a material definition that will be referenced later by its integer index, `39`.

Each property in the array expects a fixed number of floats to follow it. `DIFFUSE` 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 that represent the red, green and blue ambient and specular reflectance coefficients, respectively. All color components should be in the range 0.0 to 1.0. The system *clamps* (limits) the color components to a maximum value of 1.0, and scales the components by 255 before loading them in the framebuffer.

Specular reflectance also depends on the viewpoint. You specify the viewpoint in the lighting model. (See the description of the `LOCALVIEWER` property under Defining a Lighting Model in this Section for more details.)

`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 disables specular reflection entirely.

`EMISSION` is followed by three floats, in the range 0.0 to 1.0, that specify the red, green and blue components of the emitted light. This example defines a material that has only an emission property:

```
static float mat[] = {
   EMISSION, 0, .369, .165,
   LMNULL
};
lmdef(DEFMATERIAL, 39, 2, mat);
```

Defining a Light Source

There are two types of light sources that you can use: a *point light source* and an *infinite light source*. A point light source has a position and a direction. An infinite light source represents a light a great distance away (like the sun). Infinite light sources provide subtle overall lighting and have a performance advantage over point light sources.

The next example defines a slightly reddish-colored point light source:

```
static float lt[] = {
   LCOLOR, 1, .8, .8,
   POSITION, 0, 1.5, -.5, 1,
   LMNULL
};
lmdef(DEFLIGHT, 27, 0, lt);
```

This light source definition is associated with the integer index `27`, by which it can later be referenced.

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.

`AMBIENT` is followed by three floats, specifying the red, green and blue components of the ambient light associated with the light source.

`POSITION` is followed by four floats that are the *x*, *y*, *z*, and *w* coordinates of the light source. The *x*, *y*, and *z* coordinates represent the location of the light source in object coordinates. Note that the light source direction points from the vertex to the light source. The GL normalizes the light source direction.

Light source position is defined in homogeneous coordinates. If the *w* coordinate is zero, an infinite light source is defined; a non-zero *w* component defines a point light source and the *x*, *y*, and *z* coordinates represent the direction from the origin to the light source. This example defines 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, 3, lt);
```

Defining a Lighting Model

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, use the same syntax for a lighting model as for a material or light source. The `AMBIENT` property specifies the color of ambient light associated with the entire scene. This ambient light is nondirectional.

Specular reflectance from a point on a surface depends on the normal, the direction to the light source, and the direction to the viewpoint.

You can use a *local viewpoint* (placed at the origin) or an *infinite viewpoint* (placed at an infinite distance on the positive *z* axis).

Use `LOCALVIEWER` to indicate whether the viewpoint is local or infinite. `LOCALVIEWER` is followed by a single float, either 0.0 to indicate that the viewpoint is infinite, or 1.0 to indicate that the viewpoint is local. With a local viewpoint, the direction to the viewpoint needs to be recalculated at each vertex, which may cause a decrease in performance.

This example defines a lighting model with an infinite viewpoint:

```
static float lm[] = { LOCALVIEWER, 0, LMNULL };
lmdef(DEFLIGHT, 14, 2, lm);
```

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 surfaces are lighted with constant color, which might appear slightly unrealistic.

9.3 Binding Lighting Definitions

After defining the three components of lighting, you must *bind* (activate) them. Before binding any lighting definitions, you must be in multimatrix mode:

```
mmode(MVIEWING);
```

The next example shows how to bind the previously defined material, light source, and lighting model.

```
lmbind(MATERIAL, 39);
lmbind(LIGHT0, 27);
lmbind(LMODEL, 14);
```

Material index `39` represents the material definition from our previous example. You activate this material definition by binding it to the target `MATERIAL`. Only one material can be active at any time. Light source `27` from the previous example is activated by binding it to the target `LIGHT0`.

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 number of the light source targets can be active at any time. You can bind a light definition to only one light source target.

The current ModelView matrix transforms the position of the light source when you call `lmbind()` (see Chapter 7, “Coordinate Transformations” for more information on 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.

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 lighting models. Any index between 1 and 65535 is legal. Index 0 is used to turn lighting off, as described in Section 9.4, "Changing Lighting Settings."

9.4 Changing Lighting Settings

Use the `lmbind()` subroutine to change or deactivate (unbind) currently bound lighting definitions. Specify index 0 with `lmbind()` to unbind a material, light, or lighting model.

This example turns off the light source bound to `LIGHT0`:

```
lmbind(LIGHT0, 0);
```

Note: After this `lmbind()` call, lighting remains on because the presence of active lights is not necessary for lighting.

Recall that lighting is enabled when both a material and a lighting model are bound. To turn off lighting, you must unbind either the *material* or the *lighting model*. This example turns lighting off:

```
lmbind(LMODEL, 0);
```

Suppose you have defined more than one material for geometry you are drawing. When you bind the second material definition, it becomes the active material, overriding the previously bound material.

You can also use `lmdef()` to change the properties of an existing definition instead of creating a new definition for each variation to that definition.

Recall the point light source definition `27` from the previous example. This example changes this light source definition to produce a greenish color:

```
static float lt[] = {
   LCOLOR, .8, 1, .8,
   LMNULL
};
lmdef(DEFLIGHT, 27, 2, lt);
```

If light source definition 27 is currently bound to a light source, such as LIGHT0, the light color change takes effect immediately. Because LCOLOR is the only property in this array, only this property is changed. Changes made to bound definitions are effective immediately. Only the specified properties are changed, the other properties remain the same.

A light source position is transformed by the ModelView matrix at the time of the lmbind() call. It can be retransformed by binding it again. This is often used to make a light move from frame to frame. A light source that was previously bound can only be rebound to its original light target.

9.5 Default Settings

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[] = {
   ALPHA, 1.0,
   AMBIENT, .2, .2, .2,
   COLORINDEXES, 0, 127.5, 255,
   DIFFUSE, .8, .8, .8,
   EMISSION, 0.0, 0.0, 0.0,
   SPECULAR, 0.0, 0.0, 0.0,
   SHININESS, 0.0,
   LMNULL
};

static float lt[] = {
   AMBIENT, 0.0, 0.0, 0.0,
   LCOLOR, 1.0, 1.0, 1.0,
   POSITION, 0.0, 0.0, 1.0, 0.0,
   SPOTLIGHT, 0.0, 180.0,
   SPOTDIRECTION, 0.0, 0.0, -1.0,
   LMNULL
};
```

```
static float lm[] = {
   AMBIENT, .2, .2, .2,
   ATTENUATION, 1.0, 0.0,
   ATTENUATION2, 0.0,
   LOCALVIEWER, 0.0,
   TWOSIDE, 0.0,
   LMNULL
};
```

Passing `NULL` as the fourth parameter of an `lmdef()` call creates a definition of a default set of properties. It is also equivalent to passing an array of type `float` that has one element, the special constant `LMNULL`.

The following example sets defaults for the example lighting model:

```
lmdef(DEFMATERIAL, 39, 0, NULL);
lmdef(DEFLIGHT, 27, 0, NULL);
lmdef(DEFLMODEL, 14, 0, NULL);
```

9.6 Advanced Lighting Features

This section covers advanced lighting features. Not all lighting features are supported on all systems. Consult the `lmdef()` man page to determine which features are available on your system.

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 in the lighting model and applies to all point light sources. Attenuation does not apply to infinite or ambient light sources. Attenuation is not supported on all systems; it is ignored on systems that do not support it.

Attenuation is a function of the distance between a point light source and the surface it illuminates. You can specify *constant*, *linear* and *distance-squared* attenuation factors.

The formula for attenuation is:

attenuation factor = 1 / (k0 + k1∗dist + k2∗dist∗dist) (EQ 9-1)

where:

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.
k2	controls distance-squared attenuation.

The attenuation formula is calculated for each lighted vertex. The following example specifies constant and linear attenuation factors for lighting model `14`.

```
static float atten[] = {
   ATTENUATION, .1, 1,
   LMNULL
};
lmdef(DEFLMODEL, 14, 2, atten);
```

The `ATTENUATION` property is followed by two non-negative floats. These floats specify *k0* and *k1* of the attenuation formula.

This example adds distance-squared attenuation to lighting model `14`:

```
static float atten[] = {
   ATTENUATION, .1, 0,
   ATTENUATION2, 1,
   LMNULL
};
   lmdef(DEFLMODEL, 14, 3, atten);
```

The `ATTENUATION2` property is followed by one non-negative float, specifying *k2*.

The attenuation factor defaults are *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

A point light source is omnidirectional by default. You can make a point light source into a directional *spotlight* using the `SPOTLIGHT` property. Spotlights are available only on certain systems. See the `lmdef()` man page to determine whether your system supports spotlights.

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 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.

This example 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, 5, spot);
lmbind(LIGHT0, 11);
```

The `SPOTLIGHT` property is followed by two floats specifying the exponent and spread angle of the light cone. The exponent can range from 0 to 128; an exponent of 100 specifies a sharp falloff. The spread angle can range from 0 to 90 degrees; a spread angle of 45 defines a cone with a radius angle of 45 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. The direction vector 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.

This example turns off the spotlight effect:

```
static float spotoff[] = { SPOTLIGHT, 0.0, 180.0, LMNULL };
lmdef(DEFLIGHT, 11, 2, 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 when 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 non-linear intensity gradient across a surface. To make the approximation of the gradient as accurate as possible, place the vertices of the surface illuminated by a spotlight very close to one another.

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 correct for the front faces of those polygons. This is called one-sided lighting. With *two-sided lighting*, the lighting for backfacing polygons is also correct. Two-sided lighting is available only on certain systems. Use getgdesc(GD_LIGHTING_TWOSIDE) to determine if two-sided lighting is available.

This example adds two-sided lighting to a lighting model definition:

```
static float two[] = { TWOSIDE, 1, LMNULL };
    lmdef(DEFLMODEL, 14, 3, two);
```

The TWOSIDE property is followed by one float, either 1.0 or 0.0, 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 faces. 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", for more information about two-sided polygons.

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

When you change the properties of an active (currently bound) definition, those changes take effect immediately.

Assume that material definition `39` exists and is currently bound. The following example changes its `DIFFUSE` property immediately:

```
static float mat[] = {
   DIFFUSE, .369, 0, .165,
   LMNULL
};
   lmdef(DEFMATERIAL, 39, 2, mat);
```

This mechanism provides a general method for updating the properties of a material, a light source, or a lighting model.

It is useful to have an even more efficient method of changing material properties. For example, it is reasonable to change a specific material property at each vertex of many polygons. For this reason, the GL provides a *fast update* mechanism for material properties.

You use the `lmcolor()` subroutine to set 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()`.

Note: For higher performance, call `lmcolor()` only once, prior to drawing a large number of primitives. Do not call `lmcolor()` within a primitive (between `bgn` and `end` calls). If you must change more than one material property per vertex, excluding related `LMC_AD` ambient or diffuse changes, it is probably better to use `lmdef()` than to mix `lmcolor()` and `color()` calls within a single primitive.

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.

This example illustrates using `lmcolor()` to update 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.

This example updates 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 9.2.1, “Defining Surface Normals.” The call to `lmcolor(LMC_AD)` sets a mode where the calls to `cpack()` directly update the ambient and diffuse material properties simultaneously.

The 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. See Section 9.6.5, “Transparency,” for alpha information. 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 fast updates to material properties affect both the front and back faces. If a `BACKMATERIAL` is bound, changes to material properties affect only the front face.

9.6.5 Transparency

Normally, materials are opaque. You can control the transparency of a material with the *alpha* component. Not all systems support alpha. Alpha information is ignored by systems that do not support it. 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`. See Chapter 4 for more information on transparency.

9.6.6 Lighting Multiple GL Windows

You can use lighting in more than one GL window. The definitions that `lmdef()` creates and modifies are shared among all GL windows of a process. On the other hand, the material, light sources, and lighting model that `lmbind()` activates are specific to the GL window that was active at the time of the `lmbind()` call.

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 among the different graphics products as well as among different software releases on the same product. Nevertheless, there are some guidelines you can follow.

9.7.1 Restrictions on ModelView, Projection, and User-Defined 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; however, there are certain restrictions to be aware of.

These restrictions apply to the ModelView, Projection and user-defined transformation matrices when lighting is used:

- In some cases, normals transformed by the ModelView matrix do 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 uniform scale factors; `scale(x,y,z)` only if $x = y = z$.
- Ensure that you use an *orthonormal* (see the `lmdef()` man page for a definition of orthonormal) matrix with `loadmatrix()` or `multmatrix()`. If you use `loadmatrix()` or `multmatrix()` to specify a non-orthonormal matrix in the ModelView matrix, you must use `nmode(NNORMALIZE)` to renormalize the normals properly.
- No projection components are allowed in the ModelView matrix. More specifically, the right-most column of the matrix must be `[0 0 0 1]`.
- Two restrictions apply to the Projection matrix:
  - No rotation is allowed.
  - The top two elements of the right column must be 0.

  IRIS-4D/VGX, VGXT and SkyWriter systems permit a general 4×4 Projection matrix, so the Projection matrix restrictions do not apply to these systems.

9.7.2 Computational Considerations

Certain lighting features and operations take longer to calculate than others. The following are areas where there are performance tradeoffs:

- Two-sided lighting takes extra computation, but its performance should be better than half the performance of one-sided lighting.
- Each additional light source takes extra computation. The more you use, the longer it takes to compute the color for a vertex.
- A time-consuming 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. Any of these features adversely affects performance.
- 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. See Appendix A for subroutines that can be called between a `bgn/end` sequence.
- Calling `lmcolor()` with an argument other than `LMC_COLOR` or `LMC_NULL` might incur a slight performance penalty.

Following are suggestions for getting the highest possible performance from lighting:

- 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 drawing polygons.
- On IRIS-4D/VGX and SkyWriter systems, using 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

You can do lighting in color map mode, but that method is generally designed for systems without enough bitplanes 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 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 in color map mode uses the `COLORINDEXES` and `SHININESS` properties:

```
static float mat[] = {
   COLORINDEXES, 512, 576, 639,
   SHININESS, 5,
   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 in color map mode uses only the `POSITION` property:

```
static float lt[] = {
   POSITION, 0, 0, 1, 0,
   LMNULL
};
```

Each light source contributes an intensity proportional to its LCOLOR values, so that the scene doesn't get washed out when more that one light source is used.

IRIS Indigo systems allow local lights in color map mode. On other systems, 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.

A lighting model definition in color map mode uses only the `LOCALVIEWER` property:

```
static float lm[] = {
   LOCALVIEWER, 0,
   LMNULL
};
```

Only an infinite viewpoint is allowed. `LOCALVIEWER` can be set only to 0, which is the default.

9.9 Sample Lighting Program

The following sample program, *cylinder2.c*, demonstrates GL lighting.

```
#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, 1, 0,
   LMNULL
};

main()
{
long xorigin, yorigin, xsize, ysize;
float rx, ry;
short val;

   winopen("cylinder");
   qdevice(ESCKEY);
   getorigin(&xorigin, &yorigin);
   getsize(&xsize, &ysize);
   RGBmode();
   doublebuffer();
   gconfig();
   lsetdepth(getgdesc(GD_ZMIN), getgdesc(GD_ZMAX));
   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 (!(qtest() && qread(&val) == ESCKEY && val == 0)) {
      ry = 300 *
(2.0*(getvaluator(MOUSEX)-xorigin)/xsize-1.0);
      rx = -300 *
(2.0*(getvaluator(MOUSEY)-yorigin)/ysize-1.0);
      czclear(0x404040, getgdesc(GD_ZMAX));
      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();
    }
}
```

Chapter 10

Framebuffers and Drawing Modes

This chapter describes modes used for accessing different "layers" of your graphics scene. The subroutines described in this chapter let you draw an image on top of the standard pixel contents (*overlay*) or underneath (*underlay*). Personal IRIS and IRIS Indigo owners with the minimum bitplane configuration (eight bitplanes) can skip the sections dealing with overlay and underlay bitplanes, because these bitplanes are not present on these systems.

- Section 10.1, "Framebuffers," describes the framebuffers on the IRIS workstation.
- Section 10.2, "Drawing Modes," describes the modes used for drawing.
- Section 10.3, "Writemasks," tells you how to use masks to draw selectively into designated layers.
- Section 10.4, "Configuring Overlay and Underlay Bitplanes," tells you how to designate drawing layers for multilayer drawings.
- Section 10.5, "Cursor Techniques," tells you how to create and use cursors.

10.1 Framebuffers

Pixel data is stored in a special memory called the *framebuffer*. In its default configuration, the framebuffer on IRIS workstations is divided into four partitions that the GL can address as if they were separate framebuffers or layers of bits called *bitplanes*. These four bitplanes are called: *normal*, *overlay, underlay,* and *pop-up*. Setting a *drawing mode* selects one of these bitplanes for access.

The exact number of bits varies from system to system and is also dependent on the drawing mode. Use `getgdesc()` with the appropriate `GD_BITS` parameter to query the system for the bit configuration in each mode.

RealityEngine systems have additional framebuffers, a multisample buffer for antialiasing, and left and right buffers for stereo applications.

10.1.1 Normal Framebuffer

Usually, drawing is done in the standard (normal) color framebuffer. Data in the framebuffer is interpreted either as RGB and, optionally, Alpha, data for RGB mode, or as indices into a color map for color map mode. A single sample of the color is written to the framebuffer to represent 1 pixel.

The color represented by a pixel in the framebuffer is not necessarily the color drawn on the screen. Colors from other framebuffers can appear on top of or underneath this pixel, or colors can be blended with the pixel to achieve interesting effects such as transparency. For example, a cursor moving over a pixel obscures the pixel's color while the cursor is displayed in that location. The original color of the pixel is displayed once again after the cursor passes. Similarly, when a pop-up menu is drawn over a window, the underlying colors are temporarily obscured, but they reappear when the pop-up menu disappears. These effects are accomplished by drawing into other parts of the framebuffer such as the overlay, underlay, or pop-up planes.

On RealityEngine systems, a *multisample* buffer coexists with the normal framebuffer that provides single-pass multisample antialiasing, which is described in Chapter 15. Multisampling can be used only when the draw mode is `NORMALDRAW`, and only when you have configured the multisample buffer.

RealityEngine systems also feature a flexible framebuffer configuration that allows you to specify how to partition the framebuffer and how to allocate bits within the various parts of the framebuffer. You can configure the framebuffer to choose the number of subsamples for multisample antialiasing and you can also allocate framebuffer bits for z-buffer and stencil operations.

10.1.2 Overlay Framebuffer

Overlay planes are useful for creating menus, construction lines, rubber-banding lines, and so on. 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.

10.1.3 Underlay Framebuffer

Underlay planes are useful for background grids that appear where nothing else is drawn, such as a reference grid for a sketching application. 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 two underlay bitplanes, there are four possible underlay colors.

Use `drawmode()`, described in Section 10.2, “Drawing Modes,” to enter overlay or underlay mode and to return to normal drawing mode—drawing into the standard bitplanes.

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; you can allocate the others among overlay bitplanes, underlay bitplanes, or neither. See Section 10.4, “Configuring Overlay and Underlay Bitplanes.”

10.1.4 Pop-up Framebuffer

The two 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 not recommended.

10.1.5 Left and Right Stereo Framebuffers

This section describes a feature that is available only on RealityEngine systems, so you may want to skip to Section 10.2, “Drawing Modes,” if you do not have one of these systems.

On RealityEngine systems, the normal framebuffer can be configured for stereoscopic viewing with the `stereobuffer()` command.

When `stereobuffer()` is used in conjunction with `singlebuffer()`, two color buffers, left and right, are allocated. When `stereobuffer()` is used in conjunction with `doublebuffer()`, four color buffers, front-left, front-right, back-left, and back-right, are allocated.

`stereobuffer()` does not take effect until `gconfig()` is called. The default mode is monoscopic viewing. If neither `monobuffer()` nor `stereobuffer()` is called, `gconfig()` defaults to monoscopic buffer mode.

`stereobuffer()` configures the framebuffer to store left and right images, but it does not position those images as required for stereo viewing. Use `setmonitor()` to select a stereo video format to display the stereo images correctly. See Chapter 5 for the `setmonitor()` parameters. When `setmonitor()` is not configured for stereo display, only the left buffer is displayed.

In singlebuffer mode, `leftbuffer()` controls whether drawing is enabled in the left buffer, and `rightbuffer()` controls whether drawing is enabled in the right buffer.

In doublebuffer mode, the front-left buffer is enabled for drawing if both `frontbuffer()` and `leftbuffer()` are true, and the back-left buffer is enabled for drawing if both `backbuffer()` and `leftbuffer()` are true. Likewise, the front-right buffer is enabled for drawing if both `frontbuffer()` and `rightbuffer()` are true, and the back-right buffer is enabled if both `backbuffer()` and `rightbuffer()` are true.

`leftbuffer()` and `rightbuffer()` should be called only when the draw mode is `NORMALDRAW`, and they are ignored when the normal buffer is not configured for stereo buffering.

TRUE is the default for `leftbuffer()` and FALSE is the default for `rightbuffer()`. After `gconfig()` is called, the left buffer is enabled and the right buffer is disabled.

10.2 Drawing Modes

The drawing mode specifies which of the four layers, *normal*, *overlay*, *underlay*, or *pop-up,* is the intended destination for the bits produced by subsequent drawing and mode commands. Calls to `color()`, `getcolor()`, `getwritemask()`, `writemask()`, `mapcolor()`, and `getmcolor()` are affected by the current drawing mode.

Rather than introduce a new set of subroutines for operating on the different layers, the color map subroutines are used to affect the overlay and underlay bitplanes if the system is in overlay or underlay mode.

For example, in overlay mode, all drawing routines draw into the overlay bitplanes rather than into the standard bitplanes. 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. The routines are similarly redefined for underlay mode.

Some system resources are shared among the bitplanes, while in other cases the system maintains individual resources for each one. The pop-up, overlay, normal, and underlay planes maintain 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`

Other modes affect only the operation of the normal framebuffer. You can modify these modes only while the normal framebuffer is selected:

```
acsize
blink
cyclemap
multimap
onemap
setmap
stencil
stensize
swritemask
zbuffer
zdraw
zfunction
zsource
zwritemask
```

All other modes are shared, including matrices, viewports, graphics and character positions, lighting, and many primitive rendering options.

There is a special bitplane area reserved for cursor images. See Section 10.5, "Cursor Techniques." In cursor mode, only `mapcolor()` and `getmcolor()` perform a function; `color()`, `getcolor()`, `writemask()`, and `getwritemask()` are ignored.

drawmode

`drawmode()` sets the current drawing mode; *mode* defines the drawing mode:

```
void drawmode(long mode)
```

Drawing modes are:

`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.

The default drawing mode is `NORMALDRAW` mode, which remains set unless you change it explicitly.

getdrawmode

`getdrawmode()` returns the current drawing mode specified by `drawmode()`:

```
long getdrawmode(void)
```

Each drawing mode has its own color and *writemask*. See Section 10.3, "Writemasks," for information about writemasks. 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.

You can use Gouraud shading, that is, `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`.

Many routines that affect the operation of the standard bitplanes should not be used while in overlay or underlay drawing mode. They include `doublebuffer()` (on all except VGX, SkyWriter, and RealityEngine systems), `RGBmode()`, `zbuffer()`, and `multimap()`. VGX, SkyWriter, and RealityEngine systems support double-buffered underlay and overlay.

10.3 Writemasks

In 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 selectively limit drawing into the bitplanes. A writemask determines whether or not a new value is stored in each bitplane. A one in the writemask allows the system to store a new value in the corresponding bitplane; a zero prevents a new value from being written, so the bitplane retains its current color.

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.

10.3.1 How Writemasks Work

Figure 10-1 shows how a writemask works. In this example, the values in the first and second bits (*b1* and *b2*) do not change because their corresponding positions in the writemask are zero. All the other values change because they have ones in their corresponding positions in the writemask.

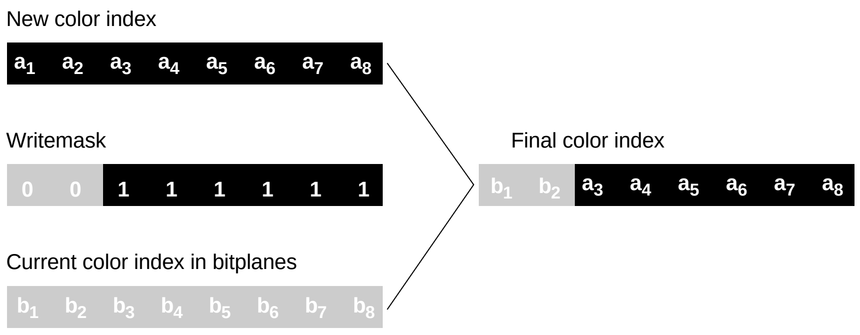

New color index
a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8
Writemask
0 0 1 1 1 1 1 1
Current color index in bitplanes
b_1 b_2 b_3 b_4 b_5 b_6 b_7 b_8
Final color index
b_1 b_2 a_3 a_4 a_5 a_6 a_7 a_8

Figure 10-1 Writemask

The writemask is described here in terms of the standard drawing bitplanes, but it works exactly the same way 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 in binary notation. 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 restricts what is written to the bitplanes. In the example above, if the writemask is 15 (bits= 00001111), only the bottom (right-most) 4 bits of the color are written into the bitplanes (1 enables writing to the bitplane; 0 disables writing to the bitplane). If the color is 68, any pixels hit by a drawing subroutine while the writemask is enabled contain ABCD0100, where ABCD are the 4 bits that were previously there. The zeros in the writemask prevent those bits from writing. The default writemask is entirely ones, so there is no restriction.

10.3.2 Writemask Subroutines

This section describes the subroutines you use to work with writemasks.

writemask

`writemask()` grants write permission to available bitplanes in color map mode:

```
void writemask(Colorindex wtm)
```

The writemask prevents writing into (protects) bitplanes in the current drawing mode that are reserved for special uses. *wtm* is a mask with 1 bit available per bitplane. Wherever there are ones 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`, the overlay bitplanes; if it is `UNDERDRAW`, the underlay bitplanes. It is 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 (see the *circles.c* sample program).

RGBwritemask

`RGBwritemask()` is the same as `writemask()`, except that it functions in RGB mode:

```
void RGBwritemask(short red, short green, short blue)
```

The arguments *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.

wmpack

`wmpack()` is the same as `RGBwritemask()`, except it that it accepts a single packed argument, rather than three separate masks:

```
wmpack(unsigned long pack)
```

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).

getwritemask

In color map mode, `getwritemask()` returns the current writemask of the current drawing mode:

```
long getwritemask(void)
```

The return value is an integer with up to 12 significant bits, one for each available bitplane.

gRGBmask

`gRGBmask()` returns the current RGB writemask as three 8-bit masks:

```
void gRGBmask(short *redm, *greenm, *bluem)
```

`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.

10.3.3 Sample Writemask Programs

Two sample programs that use writemasks follow.

The first sample program, *circles.c*, draws overlapping circles. Because of the writemask, the overlapping colors form their compound color—that is, where the red and green circles overlap, the shared area is yellow; where the red and blue circles overlap, the shared area is magenta, and so on.

```
#include <gl/gl.h>
main()
{
   prefsize(400, 400);
   winopen(“circles”);
   color(BLACK);
   clear();
   writemask(RED);
   color(RED);
   circfi(150, 250, 100);
   writemask(GREEN);
   color(GREEN);
   circfi(250, 250, 100);
   writemask(BLUE);
   color(BLUE);
   circfi(200, 150, 100);
   sleep(10);
   gexit();
   return 0;
}
```

As a more involved example, suppose you want to draw two completely independent electronic circuits on the screen: power and ground. You want the circuit to be drawn on a white background, with the power traces in blue, the ground traces in black, and short circuits (power touching ground) in red.

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*/
```

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>
writemask(1);
color(0);
clear();
```

To erase all ground circuitry:

```
writemask(2);
color(0);
clear();
```

The complete *circuit.c* sample program follows. The user 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 the mouse 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 R              0
#define G              1
#define B              2
#define RGB            3
#define BACKGROUND 0x0              /* = 00 */
```

```
#define POWER 0x1                   /* = 01 */
#define GROUND 0x2                  /* = 10 */
#define SHORT 0x3                   /* = 11 */

void powerrect(x1, y1, x2, y2)
Icoord x1, y1, x2, y2;
{
    writemask(0x1);
    color(POWER);
    sboxfi(x1, y1, x2, y2);
}

void groundrect(x1, y1, x2, y2)
Icoord x1, y1, x2, y2;
{
    writemask(0x2);
    color(GROUND);
    sboxfi(x1, y1, x2, y2);
}

void clearcircuit()
{
    writemask(0x3);
    color(BACKGROUND);
    clear();
}

main()
{
    int i, drawtype;
    Device dev;
    short val;
    short x1, y1, x2, y2;
    long xorg, yorg;
    Boolean run;
    short saved[SHORT+1][RGB];
    prefsize(400, 400);
    winopen("circuit");
    color(BLACK);
    clear();
    qdevice(PKEY);                  /* draw power rectangles */
    qdevice(GKEY);                  /* draw ground rectangles */
    qdevice(CKEY);                  /* clear screen */
    qdevice(ESCKEY);                /* quit */
    qdevice(WINQUIT);               /* quit from window manager */
    qdevice(LEFTMOUSE);             /* mark rectangle corners */
    tie(LEFTMOUSE, MOUSEX, MOUSEY);
    getorigin(&xorg, &yorg);

/* save existing color map */
    for (i = BACKGROUND; i <= SHORT; i++)
        getmcolor(i, &saved[i][R], &saved[i][G], &saved[i][B]);

/* load new color map */
    mapcolor(BACKGROUND, 0, 0, 0);              /* black */
    mapcolor(POWER, 0, 0, 255);                 /* blue */
    mapcolor(GROUND, 255, 255, 255);            /* white */
    mapcolor(SHORT, 255, 0, 0);                 /* red */
    drawtype = GROUND;
    run = TRUE;
    while (run) {
        dev = qread(&val);
        if (dev == WINQUIT)
            run = FALSE;
        else if (dev == LEFTMOUSE) {            /* downclick */
            qread(&x1);
            qread(&y1);
            qread(&val);                        /* upclick */
            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);
        }
        else if (val == 0) {/* on upstroke only */
            switch (dev) {
            case PKEY:
                drawtype = POWER;
                break;
            case GKEY:
                drawtype = GROUND;
                break;
            case CKEY:
                clearcircuit();
                break;
            case ESCKEY:
                run = FALSE;
                break;
            }
        }
    }
    /* restore default color map */
    for (i = BACKGROUND; i <= SHORT; i++)
        mapcolor(i, saved[i][R], saved[i][G], saved[i][B]);
    gexit();
    return 0;
}
```

10.4 Configuring Overlay and Underlay Bitplanes

To set the number of user-defined bitplanes 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()/underlay()` takes effect only after `gconfig()` is called, which is when all bitplane requests are resolved.

overlay

`overlay()` sets the number of user-defined bitplanes used for overlay colors:

`void overlay(long planes)`

underlay

`underlay()` sets the number of user-defined bitplanes used for underlay color:

`void underlay(long planes)`

`underlay()` is the same as `overlay()` except it affects the `underlay()` colors. The default value is 0.

planes is the number of bitplanes to use for the overlay/underlay color, which depends on the system type. The overlay/underlay framebuffer contains two bitplanes in single buffer, color map mode. Use `overlay()/underlay()` to change the number of bitplanes allocated for overlay/underlay mode. Use `drawmode()` to specify the overlay/underlay planes as the destination for drawing mode changes.

Table 10-1 shows the system type and the number of overlay/underlay planes.

System	Overlay/Underlay Planes
Personal IRIS	0 or 2 single buffer, color-map mode overlay/underlay bitplanes. (There are no overlay/underlay bitplanes in the minimum configuration of the Personal IRIS.)
IRIS-4D/G IRIS-4D/GT IRIS-4D/GTX	0, 2, or 4 single buffer, color-map mode overlay/underlay bitplanes. (Use of 4 is discouraged, because of interference with the window manager pop-up bitplanes.)
IRIS-4D/VGX, SkyWriter	0, 2, 4, or 8 single or double buffer color map mode overlay/underlay 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/underlay bitplanes with an IRIS-4D/VGX or SkyWriter system.
RealityEngine	0,2,4, or 8 single or double buffer overlay/underlay bitplanes. Does not steal alpha planes.
IRIS Indigo	0

Table 10-1 Overlay and Underlay Bitplane Configurations

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 framebuffers works.

If the overlay planes contain 0 at any location, the system displays the contents of the normal framebuffer at that location. If the underlay planes contain 0 at a given location, the system displays the color at index 0 of the underlay framebuffer's color map when the normal bitplanes do not obscure them.

This loads the color map for the underlay buffers with black, red, green, blue.

```
underlay(2); /* two bitplanes, four colors */
gconfig();
drawmode(UNDERDRAW);
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 */
```

10.5 Cursor Techniques

The cursor is handled with special cursor hardware. When the color guns scan the screen, they look at the cursor mask to determine what color to draw the cursor with as they cross the square region of the screen where the cursor is to be drawn. The cursor mask can be 1 or 2 bits deep. If the cursor mask is zero, the normal color is presented. If the mask is nonzero, the mask value is looked up in a color table (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.

10.5.1 Types of Cursors

The system supports five cursor types: a 16×16-bit cursor in one or three colors, a 32×32-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.

Default Cursor

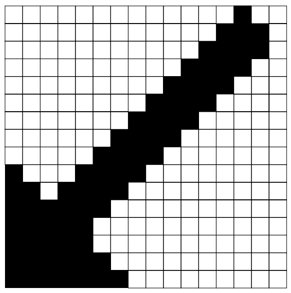
There is a default cursor, cursor number zero (0), which is an arrow pointing to the upper-left corner of the cursor glyph, and whose origin is at (0, 15), the tip of the arrow. The default cursor (number 0) cannot be redefined, and can always be used. The position of the origin of the cursor, or the cursor's hot spot, is set to the current values of the valuators that are attached to the cursor.

Cross-Hair Cursor

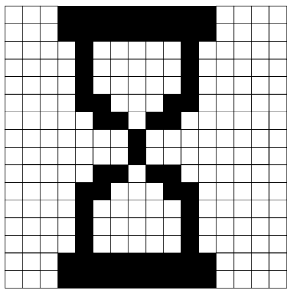
The cross-hair cursor (`CCROSS`) is formed with 1-pixel wide intersecting horizontal and vertical lines that extend completely across the screen. It is a one-color cursor that always uses cursor color 3 as its color. Its origin is at the intersection of the two lines; the default center is (15, 15). The hot spot is at the center of the cross.

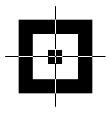
The cross-hair cursor is formed from a default glyph that cannot be changed. If you assign a value to it with `defcursor()`, the user-defined glyph is ignored. The color of the cross-hair cursor is set by mapping color index 3.

Figure 10-2 shows some example cursors.

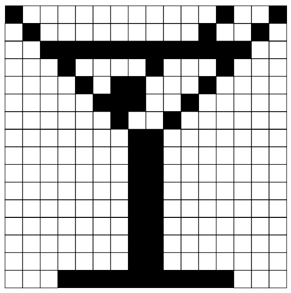


```
Cursor arrow = {0xFE00,  0xFC00,  0xF800,  0xF800,
                0xFC00,  0xDE00,  0x8F00,  0x0780,
                0x03C0,  0x01E0,  0x00F0,  0x0078,
                0x003C,  0x001E,  0x000E,  0x0004}
```





```
Cursor hourglass = {0x1FF0,  0x1FF0,  0x0820,  0x0820,
                    0x0820,  0x0C60,  0x06C0,  0x0100,
                    0x0100,  0x06C0,  0x0C60,  0x0820,
                    0x0820,  0x0820,  0x1FF0,  0x1FF0}
```



```
Cursor martini = {0x1FF8,  0x0180,  0x0180,  0x0180,
                  0x0180,  0x0180,  0x0180,  0x0180,
                  0x0180,  0x0240,  0x0720,  0x0B10,
                  0x1088,  0x3FFc,  0x4022,  0x8011}
```

Figure 10-2 Example Cursors

10.5.2 Creating and Using Cursors

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()`.

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:

$\texttt{mapcolor(1, } r_1\texttt{, } g_1\texttt{, } b_1\texttt{)}$
$\texttt{mapcolor(2, } r_2\texttt{, } g_2\texttt{, } b_2\texttt{)}$
$\texttt{mapcolor(3, } r_3\texttt{, } g_3\texttt{, } b_3\texttt{)}$

Note: Three-color cursors might not be supported on all future versions of hardware. To write code that is portable, use only single-color cursors.

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.

10.5.3 Cursor Subroutines

This section describes the cursor subroutines.

curstype

`curstype()` defines the current cursor type:

```
void curstype(long typ)
```

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.

After you call `curstype()`, call `defcursor()` to specify the appropriately sized array and to assign a numeric value to the cursor glyph.

defcursor

`defcursor()` defines a cursor glyph:

```
void defcursor(short n, Cursor curs)
```

The index *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.

curorigin

`curorigin()` sets the origin of a cursor:

```
void curorigin (short n, short xorigin, short yorigin)
```

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()`. The number *n* is an index into the cursor table created by `defcursor()`. `curorigin()` does not take effect until there a a call to `setcursor()`.

setcursor

`setcursor()` sets the cursor characteristics:

```
void setcursor(short index, Colorindex color, Colorindex wtm)
```

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 the color for the cursor with `mapcolor()` and `drawmode()`.

getcursor

`getcursor()` returns the cursor characteristics:

```
void getcursor(short *index, Colorindex *color, Colorindex *wtm, Boolean b)
```

It returns two values: the cursor glyph (*index*) and a boolean value (*b*), which indicates whether the cursor is visible.

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.

10.5.4 Sample Cursor Program

This sample program, *flag.c*, defines a three-color 32×32 cursor in the shape of a United States flag. Unfortunately, 32×32 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 <stdio.h>
#include <gl/gl.h>
#include <gl/device.h>

unsigned short curs2[128] = {
    0x0000, 0x0000, 0x0000, 0x0000,
    0x0000, 0x0000, 0x0000, 0x0000,
    0x0000, 0x0000, 0x0000, 0x0000,
    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,
    0x0000, 0xffff, 0x6666, 0xffff,
    0x6666, 0xffff, 0x0000, 0xffff,
    0x0000, 0xffff, 0x6666, 0xffff,
    0x6666, 0xffff, 0x0000, 0xffff,
    0x0000, 0xffff, 0x6666, 0xffff,
    0x6666, 0xffff, 0x0000, 0xffff,
    0x0000, 0x0000, 0x0000, 0x0000,
    0x0000, 0x0000, 0x0000, 0x0000,
    0x0000, 0x0000, 0x0000, 0x0000,
    0x0000, 0x0000, 0x0000, 0x0000,
    0xffff, 0xffff, 0xffff, 0xffff,
    0x0000, 0x0000, 0x0000, 0x0000,
    0xffff, 0xffff, 0xffff, 0xffff,
    0x0000, 0x0000, 0x0000, 0x0000,
    0xffff, 0xffff, 0xffff, 0xffff,
    0x0000, 0x0000, 0x0000, 0x0000,
    0xffff, 0xffff, 0xffff, 0xffff,
    0xffff, 0x0000, 0xffff, 0x0000,
    0xffff, 0xffff, 0xffff, 0xffff,
    0xffff, 0x0000, 0xffff, 0x0000,
    0xffff, 0xffff, 0xffff, 0xffff,
    0xffff, 0x0000, 0xffff, 0x0000
};

main()
{
    short val;
    prefsize(400, 400);
    if (getgdesc(GD_BITS_CURSOR) < 2) {
        fprintf(stderr, “2-plane cursor not available\n”);
        return 1;
    }
    winopen(“flag”);
    color(BLACK);
    clear();
    qdevice(ESCKEY);
    drawmode(CURSORDRAW);
    mapcolor(1, 255, 0, 0);
    mapcolor(2, 0, 0, 255);
    mapcolor(3, 255, 255, 255);
    drawmode(NORMALDRAW);
    curstype(C32X2);
    defcursor(1, curs2);
    setcursor(1, 0, 0);
    while (TRUE) {
    Device dev;
    dev = qread(&val);
        if (dev == ESCKEY && val == 0);
            break;
        if (dev == INPUTCHANGE &&val != 0) {
            drawmode(CURSORDRAW);
            mapcolor(1,255,0,0);
            mapcolor(2,0,0,255);
            mapcolor(3,255,255,255);
            drawmode(NORMALDRAW);
        }
    }
    gexit();
    return 0;
}
```

Chapter 11

Pixels

This chapter describes the subroutines that allow you to address screen pixels directly and perform pixel operations.

- Section 11.1, “Pixel Formats,”explains the formats used for pixel data.
- Section 11.2, “Reading and Writing Pixels Efficiently,” tells you how to access pixels.
- Section 11.3, “Using pixmode,” tells you how to customize pixel operations.
- Section 11.4, “Subimages within Images,” tells you how to access pixels that are part of larger structures.
- Section 11.5, “Packing and Unpacking Pixel Data,” explains how pixel data is compressed and decompressed.
- Section 11.6, “Order of Pixel Operations,” explains the hierarchy of pixel operations.
- Section 11.7, “Old-Style Pixel Access,” describes methods of pixel access that were used in early releases of the GL, and that are not recommended.

Pixels, like raster fonts, are not nearly as easy to transform (rotate and scale) as geometric figures. Reading and writing pixels on the screen often requires the program to get information about the window dimensions and the screen resolution.

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 a color map.

Three methods of pixel access are supported.

The first method provides a high-performance interface to a flexible set of pixel subroutines—`lrectread()`, `lrectwrite()`, `rectcopy()`, `rectzoom()`, and `pixmode()` These subroutines operate on arbitrarily sized rectangles made up of arbitrarily sized pixels. The behavior of these subroutines can be modified by setting parameters for *offset*, *stride*, *pixel size*, *zoom*, and other features described later. These functions operate in either RGB or color map mode.

The second method is compatible with older versions of the GL. It provides a high-performance interface to operate on arbitrarily sized rectangles. This set of subroutines includes `rectread()`, `rectwrite()`, and `rectzoom()`. The behavior of these subroutines can be modified only for *zoom*. These functions operate in either RGB or color map mode, but because they operate with 16-bit unsigned shorts, they are not generally useful for RGB mode.

The third method is compatible with even older versions of the GL. It provides mode-dependent subroutines to deal with, at most, one scanline at a time, which is positioned according to the system’s “current character position”. These subroutines are `readpixels()`, `readRGB()`. `writepixels()`, and `writeRGB()`. Continued use of these subroutines is not suggested because higher-performance, more flexible subroutines are available.

11.1 Pixel Formats

The following pixel formats are standard on most IRIS-4D systems:

- 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.
- 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. Signed *z*-buffer data is sign-extended to 32 bits.

- 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.

On the IRIS Indigo, and the 8-bitplane Personal IRIS, these formats are used:

- RGB values are in 8 bit packed format. The framebuffer stores 8 bits of RGB data (3 bits of red, 3 bits of green, and 2 bits of blue), but RGB pixels are written as 24 bit RGB triples. When read back, the least-significant 5 bits of red and green are zero and the least-significant 6 bits of blue are zero.
- CI values are in 8 bit (3 red + 3 green + 2 blue) packet format in single buffer mode and 4 bit (1 red + 2 green + 1 blue) in double buffer mode.
- Indigo *z*-buffer data is 32 bits in software. The 8-bitplane Personal IRIS does not come standard with a *z*-buffer; however, if you have installed it as an option, the *z*-buffer data is 24 bit, sign-extended to 32 bits.

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()`. See Section 11.3, “Using pixmode,” for information on `pixmode()` features.

11.2 Reading and Writing Pixels Efficiently

This section describes subroutines that read and write pixels with the highest possible performance. The subroutines described in this section may not be available on your system, see the man pages to find out if your system supports these commands.

Note: No color mode checking is done, so RGB data read/written in color map mode, and vice-versa, can return undesired results.

11.2.1 Pixel Addressing

Pixel coordinates on the IRIS workstation are at the center of each pixel, as mentioned in Chapters 2 and 7. Because of this, you need to specify pixel boundaries in half-steps, so pixels will be centered on integer values. If you do not do this, your pixel operations can be off by a pixel or more due to round-off errors in the floating-point calculations.

Pixel operations can occur in any of the 4 GL framebuffers (bitplanes), as selected by `drawmode()`. These subroutines establish sources and destinations for pixel operations, as well as other drawing subroutines. Sources apply to pixel reads and copies; destinations apply to pixel writes and copies.

drawmode

`drawmode()` determines the drawing mode, thereby, the destination for pixel operations. `NORMALDRAW` is the default, `OVERDRAW`, `UNDERDRAW`, and `PUPDRAW` are options. In `NORMALDRAW` mode, `frontbuffer()`, `backbuffer()`, and `zbuffer()` apply. If you assert more than one destination in `NORMALDRAW` mode, more than one destination is written.

readsource

`readsource()` determines the GL framebuffer source of pixels read by `rectread()`, `lrectread()`, `rectcopy()`, `readpixels()`, and `readRGB()`. The source depends on the current drawing mode, as set by `drawmode()`.

The default value of *src* is `SRC_AUTO`, which selects the front buffer of the current GL framebuffer in single buffer mode and the back buffer in double buffer mode. `SRC_FRONT` reads from the front buffer, and `SRC_BACK` reads from the back buffer. `SRC_ZBUFFER` reads 24-bit data (32 bit for Indigo) from the *z*-buffer. Other sources such as `SRC_FRAMEGRABBER` are available if special hardware is installed. Some `readsource()` parameters are valid only on certain models; see the `readsource()` man page for specific information about source parameters.

11.2.2 Reading Pixels

The following subroutines read screen pixels.

readdisplay

`readdisplay()` reads a rectangular screen region and returns displayed pixels in packed RGB format. `readdisplay()` returns the displayed value of each addressed pixel for all display bitplanes and modes. You specify the *x* and *y* coordinates of the rectangular region. The pixels are read into *parry* left-to-right, then bottom-to-top according to the *hints* provided. The hints are modifiers, not directives; hence, they are ignored on systems that do not support them.

Table 11-1 list the hints available and their descriptions.

Hint	Description
`RD_FREEZE`	freezes the screen by blocking all graphics calls until the read is completed. This assures that the returned data accurately represents the data displayed at the time of the call.
`RD_ALPHAONE`	returns all alpha values set to one (represented as 0xff for this command.
`RD_IGNORE_PUP`	ignores the contents of the popup framebuffer.
`RD_IGNORE_OVERLAY`	ignores the contents of the overlay framebuffer.
`RD_IGNORE_UNDERLAY`	ignores the contents of the underlay framebuffer.
`RD_OFFSCREEN`	returns an error when attempting to read beyond the framebuffer boundary.

Table 11-1 Hints for `readdisplay()`

rectread and lrectread

`rectread()` 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.

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}, that is, sarray[0]=9, sarray[1]=10, etc. `rectread()` returns the number of pixels successfully read.

Normally, the number of pixels read 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 perform 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()` 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.

11.2.3 Writing Pixels

The following subroutines write to screen pixels:

rectwrite and lrectwrite

`rectwrite()` 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, then bottom to top `rectwrite()` obeys the zoom factors set by `rectzoom()` (see `rectzoom()` below). `writemask()` and `scrmask()` apply as with other drawing primitives.

`lrectwrite()` 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()` 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()` 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 noninteger zoom, so only the integer portion of the zoom parameters apply in that case. See the `rectzoom()` man page for system dependencies.

If the following rectangle is copied after calling `rectzoom(2.0, 3.0)`:

```
1 2
3 4
```

the following copy is made; the pixel on the bottom left-hand corner is at (*new x, new y*):

```
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 sample program, *zoom.c*, 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 <gl/device.h>

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

#define ZOOM        3

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

   prefsize(400, 400);
   winopen("zoom");
   qdevice(ESCKEY);
   qdevice(TIMER0);
   noise(TIMER0, getgdesc(GD_TIMERHZ)/10);/* 10 samples/sec */
   getorigin(&org[X], &org[Y]);
   getsize(&size[X], &size[Y]);
   readsize[X] = size[X] / ZOOM;
   readsize[Y] = size[Y] / ZOOM;
   rectzoom((float)ZOOM, (float)ZOOM);
   mdev[X] = MOUSEX;
   mdev[Y] = MOUSEY;
   run = TRUE;
   while (run) {
   switch (qread(&val)) {
   case TIMER0:
      getdev(XY, mdev, mval);
      mval[X] -= org[X];
      mval[Y] -= org[Y];
      rectcopy(mval[X], mval[Y],
         mval[X] + readsize[X], mval[Y] + readsize[Y], 0, 0);
   break;
   case ESCKEY:
    if (val == 0)
      run = FALSE;
    break;
   }
 }
 gexit();
 return 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 for changing to and from RGB mode, perhaps a method to change zoom factor, and perhaps 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.

11.3 Using pixmode

`pixmode()` allows you to customize pixel operations when you use `lrectwrite()`, `lrectread()`, or `rectcopy()`. `pixmode()` is only available on certain systems, see the `pixmode()` man page for 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.

11.3.1 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, before calling `lrectwrite()` to write the pixel data, call:

```
pixmode(PM_SHIFT,8);
```

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.

11.3.2 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; before calling `lrectwrite()` call:

```
pixmode(PM_EXPAND, 1);
pixmode(PM_C0, 0x00ff0000);
pixmode(PM_C1, 0x0000ff00);
```

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.

11.3.3 Adding Pixels

You can add a constant signed 32-bit value to the least significant 24 bits of each pixel transferred by calling:

```
pixmode(PM_ADD24, value);
```

where *value* is the signed 32-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.

11.3.4 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()`.

11.3.5 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. To make the pixel transfer direction top-to-bottom, use the following command:

```
pixmode(PM_TTOB, 1);
```

Top-to-bottom mode provides faster pixel read/write performance on IRIS Indigo Entry, XS, XS24, and Elan systems than does the default.

To make the fill direction right-to-left, use:

```
pixmode(PM_RTOL, 1);
```

For instance, you can mirror a pixel rectangle that is already in the framebuffer 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);
```

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()`.

11.4 Subimages within Images

Using `pixmode()` allows you to read and write pixel subrectangles to and from a larger pixel rectangle. Suppose you have a 2000×1500 pixel image and want to work with a 100×200 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 sub-rectangles 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()`.

11.5 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.

Note: 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 3×3 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 might not begin on a word boundary. This might require using the `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 100×200. The whole image is 1250×1250, the origin is at (150,500), and the pixels are packed at 24 bits per pixel instead of 32. In this case, there are 1250 pixels at 24 bits per pixel, or 30,000 bits in each scanline. Thus there are 30,000/32 (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 sub-rectangle, 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 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 (500 ∗ 938) accounts for the first 500 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 or unpacking parameters, `PM_SIZE` or `PM_OFFSET`, has no effect on `rectcopy()`.

11.6 Order of Pixel Operations

Various `pixmode()` functions can be used together. The order of calls to `pixmode()` is of no consequence. 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.

11.7 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 `cmov()` 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 `getgdesc(GD_XMAX)`. The system paints pixels from left to right and clips them to the current screenmask.

These subroutines 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 subroutines. The behavior of `readpixels()` and `writepixels()` is undefined in RGB mode. The behaviors of `readRGB()` and `writeRGB()` are undefined in color map mode.

11.7.1 Reading Pixels

This section describes the old-style pixel reading subroutines.

readpixels

`readpixels(n, colors)` reads up to *n* pixel values from the bitplanes in color map mode, to an array of shorts, *colors*. 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 in the arrays of chars *red*, *green*, and *blue*. The values of pixels read outside the physical screen are undefined.

11.7.2 Writing Pixels

This section describes the old-style pixel writing subroutines.

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. r*ed*, *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.

Chapter 12

Picking and Selecting

This chapter discusses how to use the GL *picking* and *selecting* subroutines. Picking and selecting are used to create a “point and click” interface, where the user can position the mouse over an area of the screen and click the mouse button to choose an item on the screen.

- Section 12.1, “Picking,” describes how to test for proximity to the cursor.
- Section 12.2, “Selecting,” describes how to designate a screen region to be used for picking.

12.1 Picking

Picking identifies items on the screen that are near the cursor. When the system is in picking mode, it does not draw anything on the screen. Instead, drawing subroutines that would have been drawn near the current location of the cursor cause *hits* to be recorded in the picking buffer.

All the standard drawing routines cause hits, except raster operations, such as character strings drawn with `charstr()`. However, because `cmov()` does cause a hit, character strings can be picked because they can appear to cause hits. The cursor must be near the lower-left corner of the string for it to be picked. Because pixel subroutines such as `readpixels()` and `readRGB()` are often preceded by `cmov()`, these routines can also appear to cause hits.

To use picking effectively, your program must be structured in such a way that you can regenerate the picture on the screen whenever picking is required.

To perform picking:

1. Set the system into picking mode
2. Redraw the image on the screen using `pick()`
3. Call `endpick()`.

The results of the pick appear in a buffer specified by `pick()` and `endpick()`.

A *name stack* stores the name of the items that are hit while picking is enabled. When you call a subroutine that alters the name stack or when you exit picking mode, the contents of the name stack are copied to the buffer if a hit occurs. See Section 12.1.2, "Using the Name Stack," for more details about the name stack.

pick

`pick()` puts the system in picking mode:

```
void pick(short buffer[], long numnames)
```

The *numnames* argument specifies the maximum number of values that the picking buffer can store. The graphical items that intersect the picking region are hits and store the contents of the name stack in *buffer*.

endpick

`endpick()` takes the system out of picking mode and returns the number of hits that occurred in the picking session:

```
long endpick(short buffer[])
```

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."

12.1.1 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 each `pick()` and after each `endpick()`, to ensure that 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.

Note: 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 5 pixels high. This window runs from coordinates 1 to 4 in *x* and 1 to 5 in *y*. In order to set up a mapping between floating point coordinates and screen space (integer coordinates) that centers pixels at integer coordinates in the `ortho2()` projection, you need to specify:

```
viewport(0, 3, 0, 4);
ortho2(0.5, 4.5, 0.5, 5.5);
```

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  (x1-0.5,(float)(xsize-0.5), y1-0.5,(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 less than the corner coordinates and 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()`.

```
picksize(deltax, deltay)
```

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.)

12.1.2 Using the Name Stack

A name stack stores the name of the items that are hit while picking is enabled. The name stack is a stack of 16-bit names whose contents are controlled by `loadname()`, `pushname()`, `popname()`, and `initname()`. You can store up to 1000 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).

Note: Because the contents of the name stack are reported only when it changes, one hit is reported no matter how many drawing routines actually draw something near the cursor. If the application requires more accuracy than this, it must modify the name stack more often.

loadname

`loadname()` puts *name a*t the top of the name stack and erases what was there before:

```
void loadname(short name)
```

pushname

`pushname()` puts *name* at the top of the stack and pushes all the other names in the stack one level lower:

```
void pushname(short name)
```

Before the first `loadname()` is called, the current name is unpredictable. Calling `pushname()` before `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:

```
void popname(void)
```

initnames

`initnames()` discards all the names in the stack and leaves the stack empty:

```
void initnames(void)
```

The sample program *pick.c* 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 picking 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.

Position the cursor near one of the shapes on the screen and click the left mouse button. The results of the pick are printed to the screen. 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)"

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

```
#include <gl/device.h>
#define BUFSIZE 50

void drawit()
{
    loadname(1);
    color(BLUE);
    writemask(BLUE);
    sboxfi(20, 20, 100, 100);
    loadname(2);
    pushname(21);
        color(GREEN);
        writemask(GREEN);
        circfi(200, 200, 50);
    popname();
    pushname(22);
        color(RED);
        writemask(RED);
        circi(200, 230, 60);
    popname();
    writemask(0xfff);
}
void printhits(buffer, hits)
short buffer[];
long hits;
{
    int indx, items, h, i;
    char str[20];
    sprintf(str, "%ld hit", hits);
    charstr(str);
    if (hits != 1)
        charstr("s");
    if (hits > 0)
        charstr(": ");
    indx = 0;
    for (h = 0; h < hits; h++) {
        items = buffer[indx++];
        charstr("(");
        for (i = 0; i < items; i++) {
            if (i != 0)
                charstr(" ");
            sprintf(str, "%d", buffer[indx++]);
            charstr(str);
        }
        charstr(") ");
    }
}
```

```
main()
{
    Device dev;
    short val;
    long hits;
    long xsize, ysize;
    short buffer[BUFSIZE];
    Boolean run;
    prefsize(400, 400);
    winopen(“pick”);
    getsize(&xsize, &ysize);
    mmode(MVIEWING);
    ortho2(-0.5, xsize - 0.5, -0.5, ysize - 0.5);
    color(BLACK);
    clear();
    qdevice(LEFTMOUSE);
    qdevice(ESCKEY);
    drawit();
    run = TRUE;
    while (run) {
        dev = qread(&val);
        if (val == 0) {  /* on upstroke */
            switch (dev) {
            case LEFTMOUSE:
                pick(buffer, BUFSIZE);
                    ortho2(-0.5, xsize - 0.5, -0.5, ysize - 0.5);
                    drawit();/* no actual drawing takes place */
                hits = endpick(buffer);
                ortho2(-0.5, xsize - 0.5, -0.5, ysize - 0.5);
                color(BLACK);
                sboxfi(150, 20-getdescender(), size -1, 20 + getheight());
                color(WHITE);
                cmov2i(150, 20);
                printhits(buffer, hits);
                break;
            case ESCKEY:
                run = FALSE;
                break;
                }
        }
    }
    gexit();
    return 0;
}
```

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(short buffer[], long numnames)
```

`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 that you 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.

endselect

`endselect()` turns off selecting mode:

```
long endselect(short buffer[])
```

b*uffer* 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*.

This sample program, *select1.c*, 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, and beeps when the ship collides with the planet.

```
#include <stdio.h>
#include <gl/gl.h>
#include <gl/device.h>
#define X           0
#define Y           1
#define XY          2
#define BUFSIZE         10
#define PLANET          109
#define SHIPWIDTH       20
#define SHIPHEIGHT      10
void drawplanet()
{
    color(GREEN);
    circfi(200, 200, 20);
}
main()
{
    float ship[XY];
    long org[XY];
    long size[XY];
    Device dev;
    short val;
    Device mdev[XY];
    short mval[XY];
    long nhits;
    short buffer[BUFSIZE];
    Boolean run;
    prefsize(400, 400);
    winopen("select1");
    getorigin(&org[X], &org[Y]);
    getsize(&size[X], &size[Y]);
    mmode(MVIEWING);
    ortho2(-0.5, size[X] - 0.5, -0.5, size[Y] - 0.5);
    qdevice(LEFTMOUSE);
    qdevice(ESCKEY);
    color(BLACK);
    clear();
    mdev[X] = MOUSEX;
    mdev[Y] = MOUSEY;
    drawplanet();
    run = TRUE;
    while (run) {
    dev = qread(&val);
        if (val == 0) {  /* on upstroke */
            switch (dev) {
            case LEFTMOUSE:
                getdev(XY, mdev, mval);
                ship[X] = mval[X] - org[X];
                ship[Y] = mval[Y] - org[Y];
                color(BLUE);
                sbox(ship[X], ship[Y],
                     ship[X] + SHIPWIDTH, ship[Y] + SHIPHEIGHT);

/*  specify the selecting region to be a box surrounding the rocket ship */
                ortho2(ship[X], ship[X] + SHIPWIDTH,
                        ship[Y], ship[Y] + SHIPHEIGHT);
                initnames();
                gselect(buffer, BUFSIZE);
                    loadname(PLANET);
                    /* no actual drawing takes place */
                    drawplanet();
                nhits = endselect(buffer);

/*  restore the Projection matrix. Can't use push/popmatrix because they only
*   work for the ModelView matrix stack when in MVIEWING mode */
            ortho2(-0.5, size[X] - 0.5, -0.5, size[Y] - 0.5);

/*  check to see if PLANET was selected, nhits is NOT the number
*   of buffer elements written */
                if (nhits < 0) {
                    fprintf(stderr, "gselect buffer overflow\n");
                    run = FALSE;
                }
                else if (nhits >= 1  & buffer[0] == 1 && buffer[1] == PLANET)
                    ringbell();
                break;
            case ESCKEY:
                run = FALSE;
                break;
            }
        }
    }
    gexit();
    return 0;
}
```

Index

Tell Us About This Manual

As a user of Silicon Graphics products, you can help us to better understand your needs and to improve the quality of our documentation.

Any information that you provide will be useful. Here is a list of suggested topics:

- General impression of the document
- Omission of material that you expected to find
- Technical errors
- Relevance of the material to the job you had to do
- Quality of the printing and binding

Please send the title and part number of the document with your comments. The part number for this document is 007-1210-060.

Thank you!

Three Ways to Reach Us

- To send your comments by **electronic mail**, use either of these addresses:
  - On the Internet: techpubs@sgi.com
  - For UUCP mail (through any backbone site): *[your_site]*!sgi!techpubs
- To **fax** your comments (or annotated copies of manual pages), use this fax number: 650-932-0801
- To send your comments by **traditional mail**, use this address:

  Technical Publications
  Silicon Graphics, Inc.
  2011 North Shoreline Boulevard, M/S 535
  Mountain View, California 94043-1389